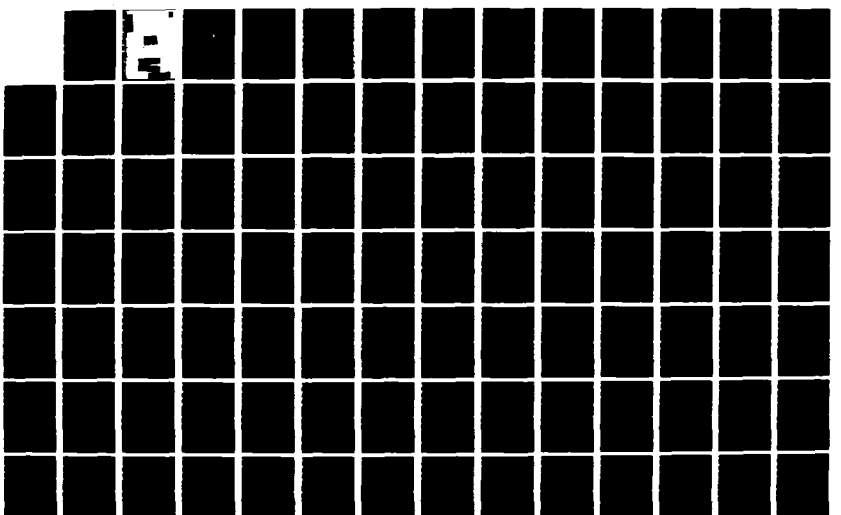
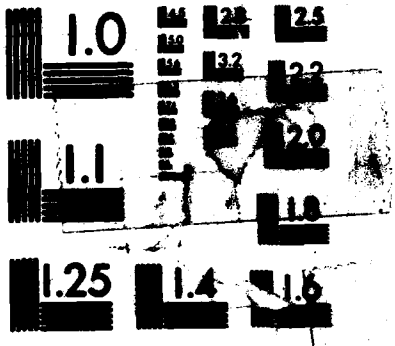
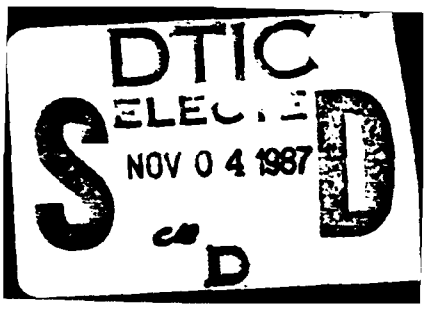
AD-A187 341 A TRAINING MANAGEMENT AND SCHEDULING SYSTEM FOR UNITED 1/2
STATES AIR FORCE T (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH M T MATTHEWS JUN 87
UNCLASSIFIED AFIT/CI/NR-87-89T F/G 5/9 NL






DTIC
ELEC
NOV 0 4 1987
S
D

A TRAINING MANAGEMENT AND SCHEDULING SYSTEM

FOR UNITED STATES AIR FORCE TACTICAL FIGHTER SQUADRONS

Mark T. Matthews, Captain, United States Air Force

A THESIS PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE IN ENGINEERING

RECOMMENDED FOR ACCEPTANCE BY
THE DEPARTMENT OF CIVIL ENGINEERING

JUNE 1987

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 87-89T	2. GOVT ACCESSION NO. AD-A187 341	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Training Management And Scheduling System For United States Air Force Tactical Fighter Squadron		5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mark T. Matthews		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: Princeton University		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433-6583		12. REPORT DATE 1987
		13. NUMBER OF PAGES 78 +
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

Lynn E. Wolaver 23 Jan 87
LYNN E. WOLAVER
Dean for Research and
Professional Development
AFIT/NR

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT

Crewmembers in United States Air Force Tactical Fighter Squadrons (TFS) accomplish a complex combination of flying and ground training to meet peacetime and wartime contingency tasking. Manual scheduling systems used today often result in crewmembers not accomplishing required training or receiving training in an inefficient manner. Flying $20 million supersonic aircraft the consequences can be expensive and fatal. The scheduling problem facing the TFS can be shown to be NP hard. A heuristic is presented which offers a solution to this scheduling problem. A series of transportation subproblems are solved using a primal network simplex code. At each stage, solutions are linked with previous solutions until a schedule is formed or no feasible solution can be found for the remaining jobs. A swap routine then attempts to find a feasible solution if one does not currently exist. The algorithm then continues into an improvement routine in an attempt to find a solution with an increased objective value. This approach was chosen due to a desire to develop a system fast enough to be interactive on a daily basis yet self contained at the squadron level. The results seem promising in providing a typical USAF TFS with training results superior to those accomplished currently .

ACKNOWLEDGEMENTS

I wish to thank the many individuals who made this work possible.

My advisor, Professor John Mulvey, was my first contact with Princeton. His early guidance led to the selection of this area of research. His patience with an Air Force fighter pilot 8 years stale of the academic environment allowed me the chance to learn. Yet his suggestions were precise enough that I did not waste unwarranted time fighting windmills in the ultimate attainment of what will hopefully develop into a practical tool for improved Tactical Fighter Squadron training.

Others also helped. Professor Warren Powell developed the primal network simplex code incorporated in this thesis. He also provided immeasurable guidance in the area of heuristic improvement procedures. Professor Michael Schneider was always available to provide help and advice in the C programming language. In addition, Hercules Vladimirou provided enormous assistance in the UNIX operating system and in modifications of the ZOOM/XMP code for incorporation in this study. Though the coding incorporated here is still somewhat crude , as my first major programming effort ever, I doubt I could have accomplished it without their assistance. I would also like to thank Professor Alain Kornhauser who, along with Professor Mulvey and Professor Powell, served on my thesis presentation committee.

This study seeks to aid real world members of a Tactical Fighter Squadron in accomplishing their training. One of these F-15 pilots, Captain Greg Feest of the 27 Tactical Fighter Squadron, took time from an intense schedule to help me in developing the guidelines of what just such a system should accomplish. Without his assistance and that of many others in the 1 Tactical Fighter Wing this study would lack the necessary detail that makes its adoption on a squadron level a real possibility.

Finally, I wish to thank my wife Donna and my son Zak for being there when I needed them. I wish to apologize to them for the times I was not there when they needed me. One can only hope that sacrifices are justified in the benefits they bring to others.

CONTENTS

LIST OF FIGURES AND TABLES

CHAPTER 3

CHAPTER 4

INTRODUCTION

A typical United States Air Force Tactical Fighter Squadron (TFS) has 40-80 crewmembers and 26 aircraft. In addition to conducting daily training missions, crewmembers must undergo intensive ground training to prepare for the multi-faceted threat which they may face and to maintain their competency of the systems on board what are increasingly complex aircraft. These tasks, in conjunction with routine administrative functions, form a tremendous burden on the crewmember's time and are difficult to schedule efficiently. The consequences of missed training can be severe. When flying at supersonic speeds, hesitating one second due to uncertainty of one's actions can mean death and the loss of a $20 million aircraft. Thus, ensuring all crewmembers are trained to accomplish their mission safely and effectively is the primary mission of the TFS.

One can describe the daily scheduling problem in a TFS as follows. Each squadron has a specified set of tasks to accomplish. Generally, the times of these tasks are fixed in advance and for practical purposes are inflexible. Within a squadron one or more pilots are qualified to perform these tasks. A particular pilot performing a particular task accrues a certain measurable benefit for either himself, the squadron, or both. Such a scheduling problem is similar to timetabling problems described in [8][17][19][27][36][46][57]. Most of these formulations assumed that the job times were a decision variable whereas in the TFS scheduling problem as in [27] jobtimes are fixed in advance. One can also describe the daily scheduling problem as a variation of the vehicle scheduling problem with multiple vehicle types or multicommodity flow problem as shown in [9].

These scheduling problems have been shown to be NP-hard [37]. Consequently most researchers have favored heuristic solution techniques such as greedy heuristics [15][49], interchange procedures [38] and heuristic partitioning [1]. Typically these heuristics are used in

conjunction with improvement procedures such as K-opt methods [7][9][39].

Other researchers have focused on exact procedures such as set partitioning [4][41][42][43][53][58] set covering [3][35][45][51] Lagrangian relaxation [23], [24][26][34][50][57] generalized networks [2][31] and network formulations [16][21][47]. (Though the problem discussed in [21] and [47] is somewhat different in structure.)

Exact procedures are impractical, however, when one has the objective of developing an interactive system which one can run on a PC. This paper presents a heuristic to schedule daily training in a tactical fighter squadron. The scheduling problem is first formulated as a transpor tation network. A primal network network simplex algorithm is then used to assign each pilot at most one job. As there are usually more jobs than pilots some jobs will remain unassigned. A new network is then formed with the remaining unassigned jobs, the pilots, and feasible arcs, where feasibility depends on the job assignments from previous iterations. An attempt is then made to assign the remaining jobs. This procedure continues until all jobs are assigned or no feasible arcs remain. If an infeasible solution results, a swapping routine attempts to find a feasible solution by swapping jobs between pilots so an unassigned job can enter the solution. Finally, an improvement routine swaps jobs between pilots in an attempt to find an increased objective function. The structure of this heuristic is somewhat analogous to that described in [27] and [7], though these two procedures incorporate a greedy and matching algorithm respectively, while the procedure outlined here will use a network simplex algorithm on a transportation network to accomplish actual job assignments. The logic used follows closely that of manual schedulers (see Appendix B) and appears promising in producing good integer solutions quickly. Furthermore this heuristic allows the flexibility to model other factors which affect training and the daily schedule other than the schedule itself. In comparison with other models, this approach appears to offer the greatest advantages in terms of performance and practicality [23].

Chapter 1 describes the structure of a typical TFS and the nature of the scheduling and training problem. Chapter 2 shows the problem formulation ands its modifications to transform it into a transportation network. In Chapter 3 results for a series of restricted size test problems are presented and compared against the results of an integer programming code. Chapter 4 discusses how one might use the model in an interactive environment to develop short and long range schedules. Finally, a summary and conclusion with recommendations for further research is presented.

CHAPTER 1

THE TFS SCHEDULING PROBLEM AND RESEARCH OBJECTIVES

1.1 The TFS Mission

The training a TFS accomplishes is specifically based on their tasking under peacetime and wartime operational or contingency plans. The most common missions are air superiority (protecting friendly forces from enemy aircraft) or ground attack (destroying enemy forces on the ground). Many units concentrate training in one area, such as F-15s training only for the air superiority mission. Others have a mixed tasking such as an F-16 unit which may devote 60% of its training towards ground attack and 40% towards air superiority. The heuristic developed here is based on the requirements for a typical F-15 TFS and its pilots though the model can easily be adopted to other organizations attempting to assign personnel to fixed time jobs.

In addition to flying training, pilots undergo training in simulators and formal classroom refresher training on all aspects of the aircraft systems, performance, and tactics as well as enemy weapon systems, performance and tactics. Other training includes such areas as survival, security, social awareness, professional military education, and post graduate work in an officer's particular area of expertise.

1.2 Duties

In addition to training, pilots accomplish specific additional duties. In an F-15 TFS these duties are in one of the following areas:

1. Weapons and tactics: Ensures squadron members understand the operation and employment of both their own aircraft, weapons, tactics and possible enemy aircraft, weapons, and tactics.

2. Operational plans: Maintains squadron's peacetime and wartime contingency plans as well as ensuring squadron members are aware of their tasking under those plans.

3. Training: Administers the flying and ground training program for new pilots (upgrade training to combat ready status) and combat ready pilots (proficiency maintaining). Training guidance comes from both higher headquarters (above squadron level) and from squadron supervisors.

4. Scheduling: Schedules pilot tasking for both flying and ground duties.

5. Standards and evaluation: Ensures the squadron and its pilots meet required proficiency through a system of ground and flying evaluations as well as routine inspections.

The following tables adapted from Air Force Manual 51-50 [56] (also see Appendix C) give a representative listing of the typical duties one would find in an Air Superiority squadron. Sorties are actual flights which range in length from 1-3 hours. With required prebriefing, post flight debriefings, and mission preparation a typical sortie requires 6-8 hours of a pilot's time. Events are specific task which occur during a sortie. The number and type of events which occur during a sortie are usually determined by the pilot's themselves though some events are specifically scheduled to occur during a given sortie. In addition some events and sorties have currency requirements. Schedulers assign these events and sorties to ensure a pilot does not become non-current.

TABLE 1.1

Representative List of Typical TFS Scheduling Requirements*

Flying

Air Combat Training (ACBT) Sorties	Collateral Sorties
Advanced Handling	Cross Country
Basic Fighter Maneuvers	Instrument Training
Air Combat Maneuvers	Mission Support
Air Combat Training	Check Flights
Dissimilar Air Combat Training	Confidence Flights
Day Intercept Missions	
Night Intercept Missions	
Specific Events	**Currency Requirements**
Day Aerial Refueling	ACBT
Night Aerial Refueling	Landing
Low Level	Night Landing
Aerial Gunnery	Formation Takeoff
	Formation Landings
	Wing Takeoffs

* events not shown are not scheduled but are accomplished during scheduled missions

TABLE 1.2

Ground Training

Weapons and Tactics	Evaluations
22 different subjects	Air Combat Test
Gun Camera Film Review	Instrument Refresher
Life Support/Survival	Plans/Other
Egress	4 Briefings
Ejection	Security
Ground Survival	Social Actions
Water Survival	Small Firearms
Theater Survival	Dental Physical
Altitude Chamber	Scheduled Exercises

TABLE 1.3

Other Scheduled Events

Meetings	Irregular
Daily Standup	Temporary Duty
D.O. Weekly	Special Training
Schedulers Weekly	Combat Turn
Flight Com. Weekly	Fire Fighter
Flight Weekly	Static Displays
Pilot Weekly	Leave (30 days/yr)

Specific levels of accomplishment are specified in Air Force Manual 51-50 by Headquarters USAF, the major air command (of which there are four), and the wings in the major air command (of which their are approximately 36). There are usually three squadrons in each wing. These levels are usually specified for a specific semiannual period. Thus the squadron organizes its activities on a semiannual cycle.

1.3 Objectives

As stated earlier accomplishment of all training requirements is a substantial scheduling task. As most squadrons schedule by "hand" sometimes tasked training is not accomplished. In addition scheduling is inefficient and large disparities can arise in the specific training a pilot may accomplish. Thus the research goals are:

1. To develop a scheduling system for a typical TFS which will correct these deficiencies.
2. To develop a scheduling system which will be both user friendly and compatible for adoption on a Personal Computer (PC). The use of a PC is important due to the lower computing cost, the lack of timely access to a mainframe computer by squadron schedulers, and the ability to carry the PC with them when a squadron deploys to a remote location.
3. To develop a system which offers solutions to daily schedules fast enough to work in an interactive role with squadron schedulers. Due to dynamic factors such as weather and maintenance problems, scheduling inputs can change on short notice requiring quick solutions. Also, given the large number of inputs into a schedule, many of which are difficult to model without large increases in complexity (and execution times), an interactive approach is critical in determining an acceptable solution.

In developing the scheduling system the objective chosen is the maximization of training benefits to the squadron. This best meets the squadron needs for two reasons:

1. Flying time, the number of missions flown, fuel costs, and other variables, are usually fixed above the squadron level. In addition, these levels are fixed based on many exogenous factors such as Congressional budgets and the expected reliability of unit aircraft.
2. Maximizing the benefits gained from training are generally the stated goals of the squadron.

These benefits are measured both directly and indirectly. As stated earlier, specific number of training events are directed by AFM 51-50. In addition commanders have a certain degree of flexibility as to which type of training they accomplish. For example, AFM 51-50 specifies no specific number of Basic Fighter Maneuvers (BFM) sorties but instead specifies a specific number of Air Combat Training (ACBT) sorties of which BFM is one type. A commander may specify what percentage of ACBT sorties will be BFM based on his own judgement. As shown later hopefully supervisors can translate these training goals and requirements directly into scheduling outputs.

CHAPTER 2

MODEL FORMULATION

2.1 The Daily Schedule

A graph $G(N,A)$ consist of a set N of nodes and a set A of unordered pairs of nodes called arcs. The arc $i-j \in A$ with $i,j \in N$ implies a direction of flow from node i to node j. If each arc in A has a number associated with it, such as a price per unit flow over the arc, the graph is termed a network [33].

Consider the scheduling problem where one has a set of pilots I who are available to perform a set of required jobs J. For the purposes of this paper a job j is considered to be the two-tuple of a given task type and time span of the task occurence. For example task type α may start at 0600 and last 3 hours until 0900. Call this job A. Job B is also a task type α, however it starts at 0800 and, as it is also a task type α, last 3 hours until 1100.

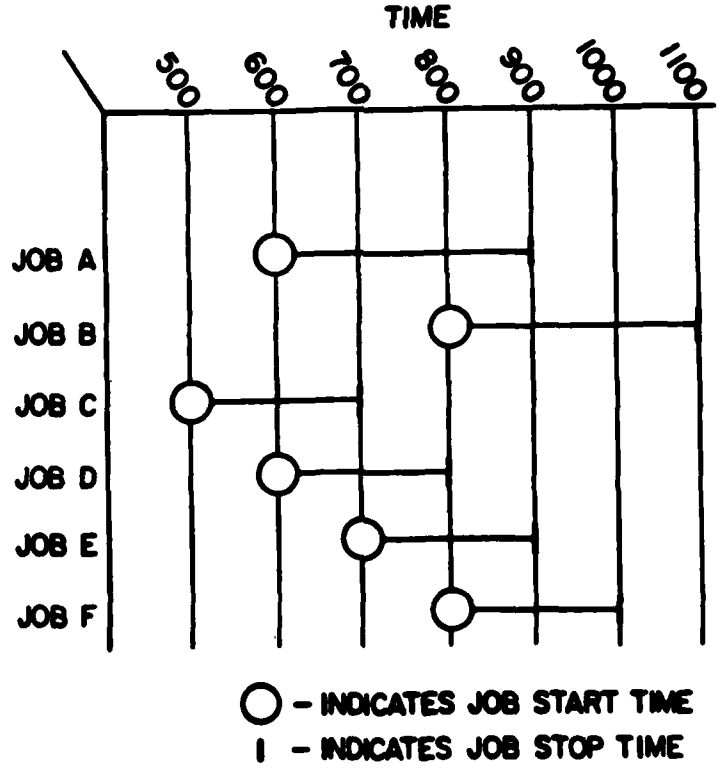


FIGURE 2.1. Task type/Task time Two-tuple representation of Jobs

Notice in Figure 2.1 that jobs A and B overlap timeperiods. Thus one individual could not perform both jobs.

For the scheduling problem presented here each pilot $i \in I$ is qualified to perform some subset of the jobs $j \in J$. One can show these qualifications in a network such as the one depicted in Figure 2.2. This is a bipartite network representation of the scheduling problem. Bipartite means that one can separate the nodes into a left or right group (here pilots are on the left and jobs on the right). Each directed arc from pilot i to job j indicates that pilot i is qualified to perform job j. The number above the arc represents a measure of the "Benefit" of pilot i performing job j. This benefit may be thought of as a price received for each unit flow across an arc from i to j. Chapter 4 discusses how this benefit is determined.

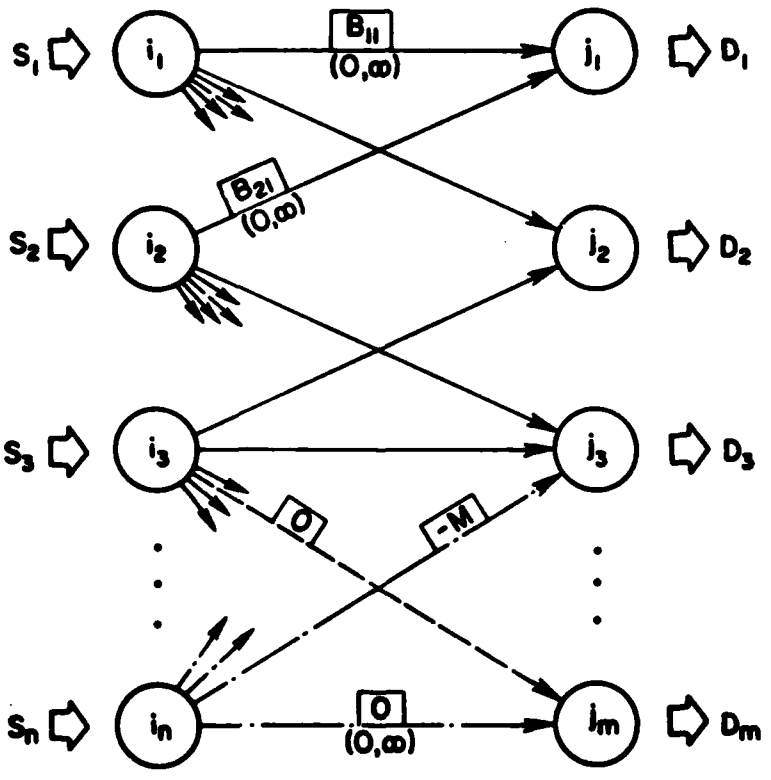


FIGURE 2.2. Bipartite Transportation Network

To understand what is meant by flow note that S_i , $i = 1,...,n$ represents how many pilots of "type" i exist. Since each pilot i is a unique individual all S_i, $i < n$ are equal to 1. D_j represents the number of jobs j. D_j is some integer value ≥ 1. Note that one could have two or more of the same task-types occurring at the same time. The S_i represent the supply or input into the left side of the network while the D_j represent the demand or output on the right hand side.

Stated another way the $\sum_{j=1}^{n} D_j$ jobs demand the services of $\sum_{i=1}^{m} S_i$ pilots. Thus the supply of pilots flows across the arcs to meet the job demands. The two numbers below the arcs indicate the minimum flow allowed across each arc (here 0 for all arcs) and the maximum allowed (infinity for all arcs).

The situation often arises where one may have more jobs than pilots. Thus pilot $i=m$ is designated a "Bogus" or dummy pilot to handle excess jobs. Conversely one may have more pilots than jobs thus one has a dummy job or sink for these pilots to perform which is designated job $j=n$. Consequently, to maintain supply equal to demand one has

$$S_m = \sum_{j=1}^{n-1} D_j$$

and

$$D_n = \sum_{i=1}^{m-1} S_i$$

The objective sought is to have flow move across the arcs in such a way that one gains the maximum total benefit. Therefore this network formulation may be represented as the following optimization problem:

$$Maximize \quad \sum_{i=1}^{m} \sum_{j=1}^{n} B_{ij} \cdot P_{ij} \tag{1}$$

$$st \quad \sum_{j=1}^{n} P_{ij} = S_i \tag{2}$$

$$\sum_{i=1}^{m} P_{ij} = D_j \tag{3}$$

$$P_{ij} \geq 0 \text{ and integer} \tag{4}$$

where

I = Set of Pilots $\{i=1,2,3,...,m\}$

J = Set of Jobs $\{j=1,2,3,...,n\}$

D_j = Job j's demand

S_i = Pilot i's supply

B_{ij} = Benefit (price) of pilot i performing job j

P_{ij} = Pilot i performing job j

The above formulation is an example of the classical network transportation problem [33]. Network theory proves that the optimal solution to the above problem will have an integer solution since all supply, demands, and arc bounds are integer [11][13][33]. Various algorithms exist which can efficiently find the solution to such problems. One of the fastest methods is the primal network simplex algorithm [12]. A brief description of how the algorithm works appears in Chapter 3. For a more detailed explanation of the algorithm and computer implementation the reader should refer to [13][20][28][29][52]. Note that the solution to the network problem as stated will in essence assign each pilot to one job. Thus one avoids the problem mentioned earlier of two jobs overlapping and the infeasibility of assigning one pilot to do both jobs. Consequently this restriction is not explicitly stated in the transportation problem as it is implicitly enforced.

However jobs will be assigned to the bogus pilot if

$$\sum_{j=1}^{n-1} D_j > \sum_{i=1}^{m-1} S_i$$

Even if there are enough pilots to cover all jobs one may have jobs assigned to "Bogus" as one has no guarantee that pilots exists who are qualified to perform any job given. This situation would mean no feasible solution exist. For the purposes of this paper, unless stated otherwise, a feasible solution is assumed to exist. However, this feasible solution may require pilots to perform more than one job.

The following example helps to illustrate this point :

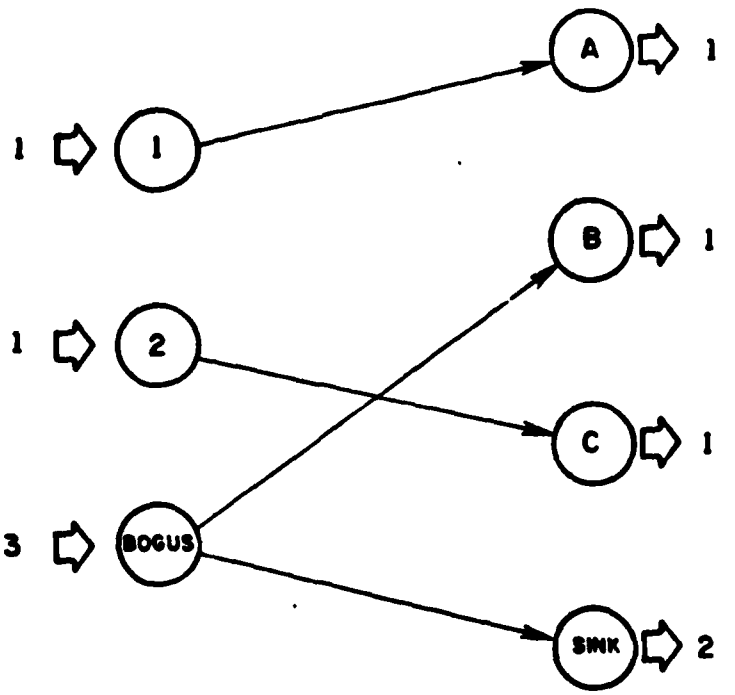


FIGURE 2.3.1. An Initial Solution to a Scheduling Problem (Only arcs with flow are shown)

From the initial transportation problem pilot 1 is assigned job A, pilot 2 job C, and "bogus" pilot 3 job B. As jobs A and C are assigned to real pilots one is left with a new assignment problem, namely to assign job B to a real pilot. Say jobs A and B have the start and stop times shown in Figure 2.2. Since Job A starts at 0800 and ends at 1100 while Job B starts at 0900 and ends at 1200 pilot 1 cannot perform Job B. Consequently one has the following new network structure based on the previous job assignments.

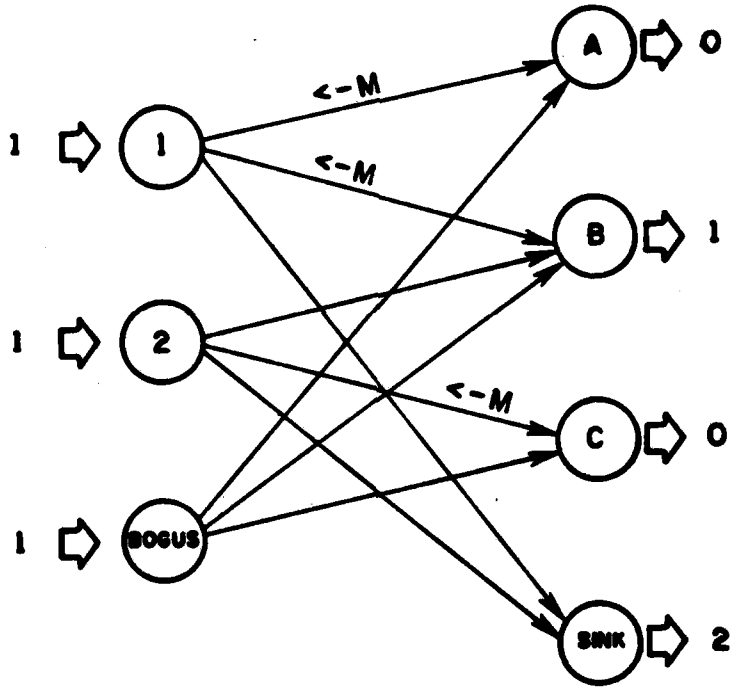


FIGURE 2.3.2. New Network Structure Following Initial Assignments

Arc cost are only shown on those arcs which are now infeasible based on the initial job assignments. Notice that since jobs A and C have been assigned their demand has dropped to zero. For each pilot i, if performing job j would conflict with a job k assigned to pilot i, then the benefit price of that pilot performing job j is changed to a value less than $-M$ where $M \rightarrow \infty$. Notice that the Bogus pilot arc prices are always $-M$. Thus a pilot with an arc price $< -M$ is never assigned to the job that such an arc points to. Consequently, the optimal solution of the network in Figure 2.3.2 is

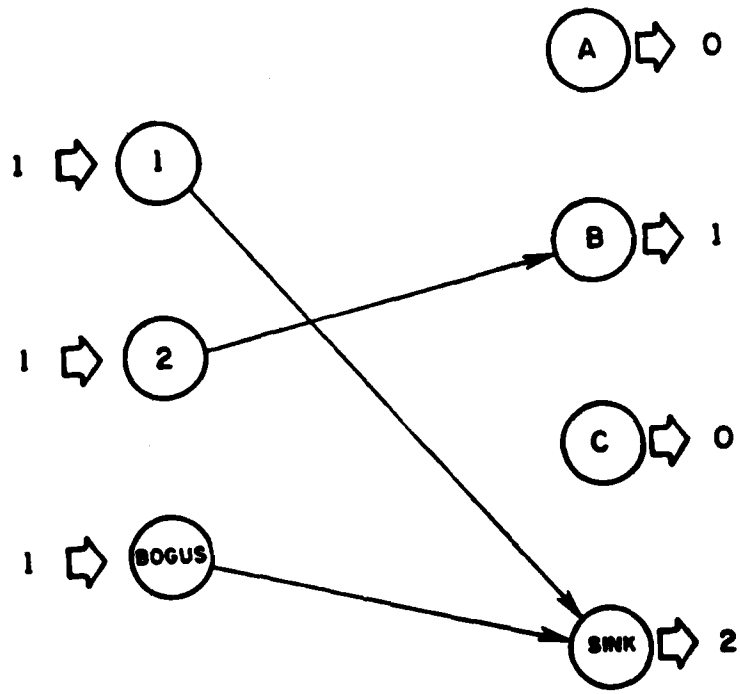


FIGURE 2.3.3. Final Solution

In summary, to assign jobs assigned to the "Bogus" pilot m from the first iteration the problem is restarted with these ''Bogus'' assigned jobs. All infeasible links have their arc price set to $< -M$ where $M \rightarrow \infty$. Infeasible links are the P_{ik} such that for a given $P_{ij} \neq 0$, $j \neq n$, $k \in J_j'$ where J_j' is the set of jobs including j which overlap the time periods job j occurs. In other words, if a pilot is busy performing job j during the time job k occurs then he cannot perform job k.

Thus on the second and subsequent major iterations the following reduced problem is solved.

$$Maximize \sum_{i=1}^{m} \sum_{j=1}^{n} B_{ij}^{rem} \cdot P_{ij} \quad \text{for all } j \in J_{rem} \tag{5}$$

$$st \sum_{i=1}^{n} P_{ij} = S_i \quad \text{for all } j \in J_{rem} \tag{6}$$

$$\sum_{i=1}^{m} P_{ij} = D_j^{rem} \quad \text{for all } j \in J_{rem} \tag{7}$$

$$P_{ij} + P_{ik} = 0 \text{ or } 1 \text{ for all } k \in J_j' , k \neq j , j \neq n \tag{8}$$

$$P_{ij} \text{ integer} \geq 0 \text{ for all } j \in J \tag{9}$$

J_{rem} = set of unassigned jobs $J_{rem} \subset J$

D_j^{rem} = updated demand for job $j \in J_{rem}$

B_{ij}^{rem} = updated benefit of pilot i doing job $j \in J_{rem}$

The algorithm is summarized as follows:

ALGORITHM A

STEP 0. Input pilot qualifications and jobs.

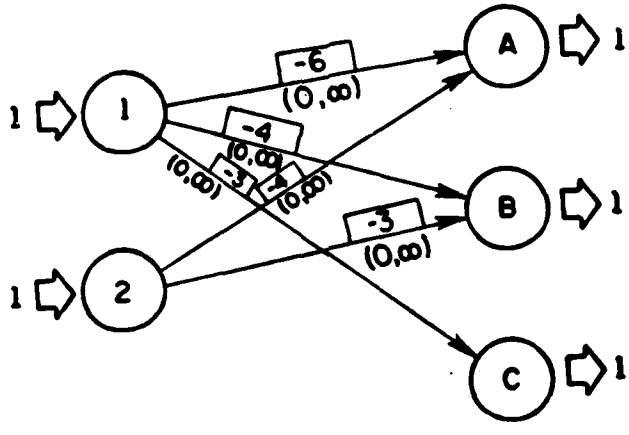
STEP 1. Assign each $m - 1$ pilot at most one job. If all jobs assigned or no feasible arcs exist for unassigned jobs go to step 3.

STEP 2. Determine infeasible arcs from assignments made in step 1. Change cost on these arcs to -M. Update job demands and benefit prices based on previous job assignments. Return to step 1.

STEP 3. Print schedule.

The cycles recur until all jobs are assigned or no feasible arcs remain. The nature of the typical daily flying schedule most often restricts the number of iterations to two with rarely more than three iterations occurring before all jobs are filled or an infeasibility occurs.

Infeasibilities occur when on subsequent iterations no pilots are available to perform one or more available jobs. With 24 flying jobs and 10-16 ancillary positions to fill there are generally enough pilots available to "fill" a schedule. However when a job exists with few pilots qualified to do it (i.e., few entering arcs to the *j* node) infeasibilities can occur. The following example illustrates this point. The dummy supply and demand nodes have been omitted. The table at the bottom of the figure gives the start and stop times of jobs A,B, and C.



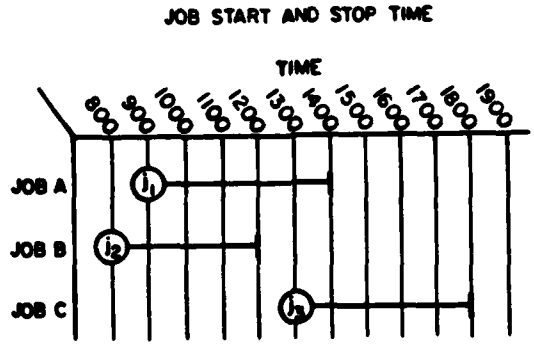


FIGURE 2.4. Network Structure of the Problem

In this example Pilot 1 is qualified to perform all three jobs while Pilot 2 is only qualified to perform jobs A and B. On the first iteration Pilots 1 and 2 are assigned jobs A and B respectively. Job C remains unassigned so a second iteration begins. However, no feasible solution exist since Pilot 1 doing job A is busy when job C begins and pilot 2 is not qualified to perform the job. Had pilot 1 been assigned job B instead of A on the first iteration, a feasible solution would exist. Several approaches to avoid such infeasibilities exist.

2.2. Handling Infeasibilities

2.2.1. Fillindexes

First one could restructure the price of each pilot performing a particular job to steer a pilot towards a feasible schedule. By dividing the number of pilots qualified to perform a job by the number of jobs, one gets an average number of pilots qualified to perform each job. This is similar to the supply demand ratios used to rank order time categories by Glassey and Mizrach [27]. Call this the fillindex of job j. Note that having 10 jobs with 10 pilots qualified to perform the job gives the same fillindex as having one pilot qualified to do one existing job. However one has greater flexibility in assigning the 10 jobs since undoubtedly some pilots can perform the same type of job twice due to time staggering between jobs. Thus one also needs a measure of how many jobs a pilot is precluded from doing if assigned job j and how hard (fillindex) the jobs will be to fill that he is precluded from doing. Thus one might use a price as follows:

$$B'_{ij} = (\omega_1 \cdot B_{ij}) + \left\lceil \frac{\omega_2 \cdot 1}{\sum_k \frac{1}{\text{fillindex}_k}} \text{ for all } k \in J'_j \cap I_i \right\rceil$$

where I_i is the set of jobs pilot i is qualified to perform.

B_{ij} above is a price structure based solely on "benefit". Here $\omega_1 + \omega_2 = 1$. With this price structure and $\omega_2 > 0$ a pilot is less likely to perform a job which interferes with a large number of other jobs he is qualified to perform. This is especially pertinent to jobs for which there exist a

large number of other pilots qualified to perform it. Setting the weights ω_1 and ω_2 is difficult, however. $\omega_1 = 1$ and $\omega_2 = 0$ gives full weight to the benefit function while $\omega_1 = 0$ and $\omega_2 = 1$ gives full weight to scheduling feasibility.

In testing of several schedules no discernible pattern arose which could consistently guarantee that infeasible schedules would not arrive regardless of the exact values of ω_1 and ω_2. Furthermore arbitrarily setting $\omega_2 > 1$ assumed some degree of infeasibility existed which is generally not the case. In setting $\omega_2 > 1$ one would sacrifice some degree of optimality to achieve feasibility even though not required. To resolve this problem one could set $\omega_1 = 1$ and $\omega_2 = 0$ and attempt to solve the problem. If a feasible solution is not found the problem can be restarted with $\omega_1 < 1$ and $\omega_2 > 0$ and rerun. One could go so far as to program a stepped increased in ω_2 (with a concurrent decrease in ω_1) until a feasible solution was found or until $\omega_1 = 0$ and $\omega_2 = 1$. However this would be unduly time consuming and one would in the end still have no guarantee of feasibility. In fact it is relatively easy to develop schedules for which the proposed fillindexes fail to achieve feasible solutions.

2.2.2. Natural Structure

A more intuitive approach incorporates the natural structuring of a daily flying schedule to avoid infeasibilities. When flying 24 sorties in a day squadrons do not fly 24 different aircraft. Takeoffs are grouped in "go's" . For example 10 aircraft may launch in the morning "go". Eight of these aircraft will launch in the midday "go" with two of the morning aircraft acting as spares. Then there may be a third "go" with six aircraft launching and four acting as spares. With such groupings pilots are generally available to fly in the third "go" after debriefing their missions from the first "go". Other scheduled ancillary duties such as supervisor of flying, squadron supervisors, and mandatory meetings, lack this natural "grouping" in scheduled time. Characteristically individuals who perform these ancillary jobs are the same individuals , such as

instructor pilots (IP) and flight examiners (SEFE), who are qualified to perform jobs with relatively low fillindexes . One could accomplish much the same affect as desired with the pricing scheme by assigning these ancillary jobs first, then assigning the flying jobs. This avoids situations where the pricing scheme does not capture scheduling hindrances that may exist. For example a schedule may have 10 basic pilot jobs to fill with 10 pilots qualified to perform these jobs giving a fillindex of one to each job. In addition one IP job and one SEFE job may exist with one of the 10 pilots qualified to perform both of these jobs. Therefore both the IP and SEFE job would have a fillindex of one. Depending on the exact times the jobs occur, the B'_{ij} derived in section 2.2.1 may or may not steer this uniquely qualified pilot into these two jobs before he is assigned to a pilot job which interferes with his accomplishing the SEFE and IP job. However, by segregating the SEFE and IP jobs and assigning them first, 9 pilots would be left to fill the 10 pilot jobs with a high likelihood, due to the natural grouping of these jobs, that a feasible schedule, flying one pilot twice, could quickly be found.

Such clusterings are similar to the partitionings outlined in [7]. However here one is establishing clusters based on empirical observations about the general characteristics of the assignment process as opposed to objective criteria such as the actual start and length of particular jobs. In an earlier formulation of the daily scheduler algorithm such clusterings of jobs was attempted. At $t = n$ all jobs that started at that time were assigned to pilots i who were qualified and availble where availability was based on jobs assigned to pilots i from $t = 1$ to $t = n - 1$. Typically there were at most four jobs with the same start times with the most common number being two. This generally resulted in 10 to 15 clusterings with each clustering requiring the initialization and solving of a transportation (albeit small) problem. This strategy tended to be unduly time consuming. With this procedure there also tended to be a higher rate of infeasibility since only jobs starting at a discrete time period were considered. Thus towards the end of the day individuals who were uniquely qualified or were one of a few who were qualified to perform a job were often

unavailable to be scheduled. Finally, this procedure resulted in a poorer overall objective value, as one might expect, since one is in essence optimizing over several small subsets of the same problem and adding the results. In the procedure finally adopted a more global approach is taken since in general one to four clusters only are employed.

For feasibility purposes the two basic clusters used were:

1. Essential ground jobs.

2. All other jobs.

Other groupings can be made that make restrictions such as no two nonflying jobs of the same type on the same day and pairings easier to manage. This point will be discussed in greater detail later.

Thus the revised heuristic may proceed as follows:

ALGORITHM B

STEP 0. Same as ALGORITHM A. In addition set $c = 1$ where $c \in C$, C = Set of job categories , $\{c = 1,....,d\}$.

STEP 1. Assign category c jobs.

STEP 2. Determine if all category c jobs are assigned or if category c jobs remain but there are no feasible arcs to these jobs. If so and $c \neq d$, set $c = c + 1$ else goto step 4.

STEP 3. Determine the infeasible arcs and update prices and demands as before. Return to step 1.

STEP 4. Print schedule.

This approach was superior to the fillindexes in most cases in its ability to resolve infeasibilities. However, again, in most cases infeasibilities would not exist under the basic algorithm. To restrict the problem to solving job assignments by categories may unnecessarily increase solution times. Conversely one still has no guarantee of a feasible solution.

2.2.3. Swapping

In a final attempt to resolve infeasibilities one could instead or in conjunction with the above procedure use a swap routine in an attempt to produce a quick feasible solution. The routine starts if one enters step 4 of Algorithm B with unassigned jobs. The swap routine uses a pointer to indicate which pilots are qualified to perform the unassigned job(s). Another pointer tracks the remaining feasible arcs. If one or more feasible arcs exist for each conflicting job assigned to a pilot who is qualified to perform the unassigned job, the unassigned job is assigned

to this pilot. The conflicting jobs which were assigned to this pilot are "unassigned" and thrown back into the job pool. Another simplex iteration is then started where, since feasible arcs to these jobs exist, these newly unassigned jobs are filled by other pilots. If one cannot find a pilot for which there are feasible arcs to all of his conflicting jobs (those jobs he is assigned which prevent him from performing the unassigned job) one assigns the unassigned job to the pilot with the lowest number of conflicting jobs. In case of ties assign the unassigned job to the pilot who has the highest B_{ij}.

Once a swap has been made the jobs swapped to a pilot are permanently assigned to him. In this way one prevents cycling in later iterations and thus guarantees termination of the algorithm although one may still terminate with an infeasible solution. This situation would most likely occur after several iterations of assigning unassigned jobs to pilots through the swap routine. Thus in the latter iterations relatively few pilots exist who can swap out assigned jobs to pick up an unassigned job. However this situation has yet to occur with testing of real world schedules.

Consider the jobs A through F with start and stop times shown in Figure 2.5.1. Based on the network structure (not shown) pilot 1 was assigned job A, pilot 2 job C, and pilot 3 job D on the first iteration.

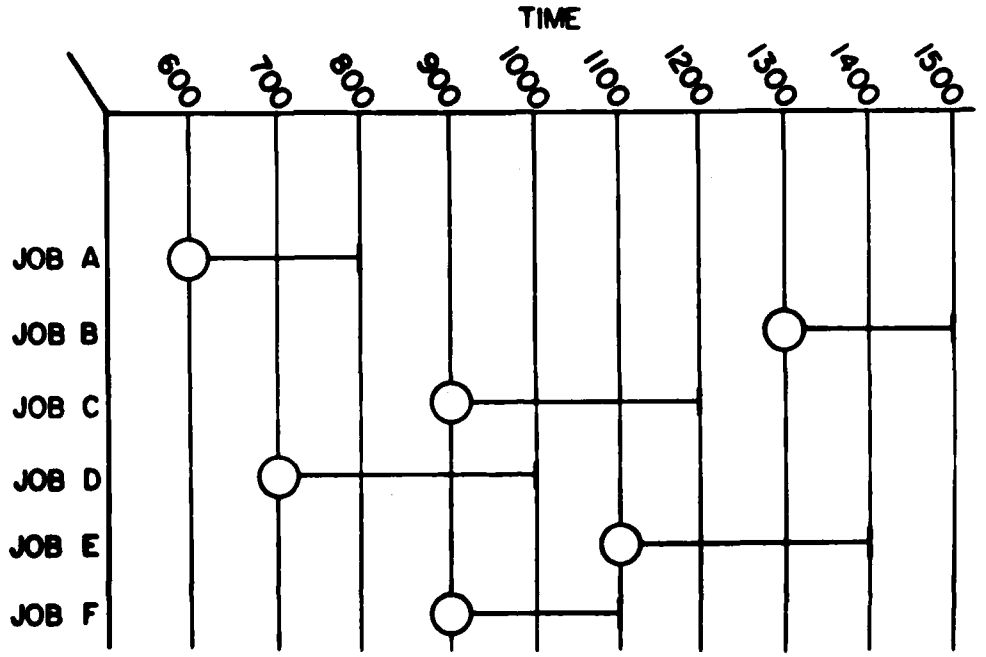


FIGURE 2.5.1. Job Start and Stop Times

On the second iteration pilot 1 was assigned job B. No further assignments were made as no feasible arcs remained. The resulting schedule is shown in Figure 2.5.2 with jobs E and F assigned to Bogus. Since no feasible arcs to these jobs exist the swap routine starts.

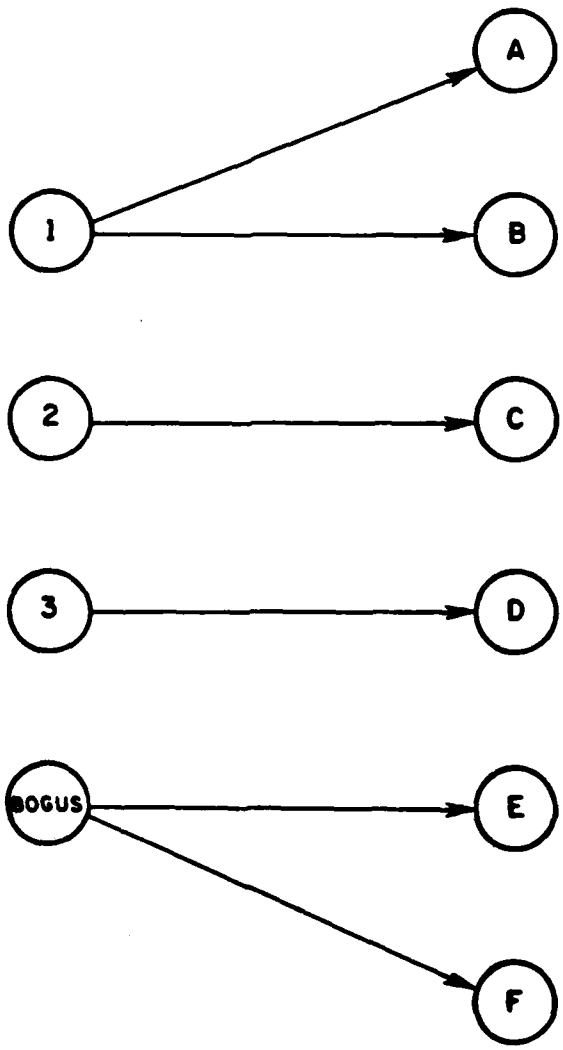


FIGURE 2.5.2. Initial Solution (sink node and flows not shown)

Pilots 1 and 2 are qualified to perform Bogus job E however pilot 1's job B and pilot 2's job C interfere with job E. Scanning the feasible arcs we find that pilot 2 can perform job B and C. Thus pilot 1 drops job B which pilot 2 picks up freeing pilot 1 to pickup job E. The new assignments are:

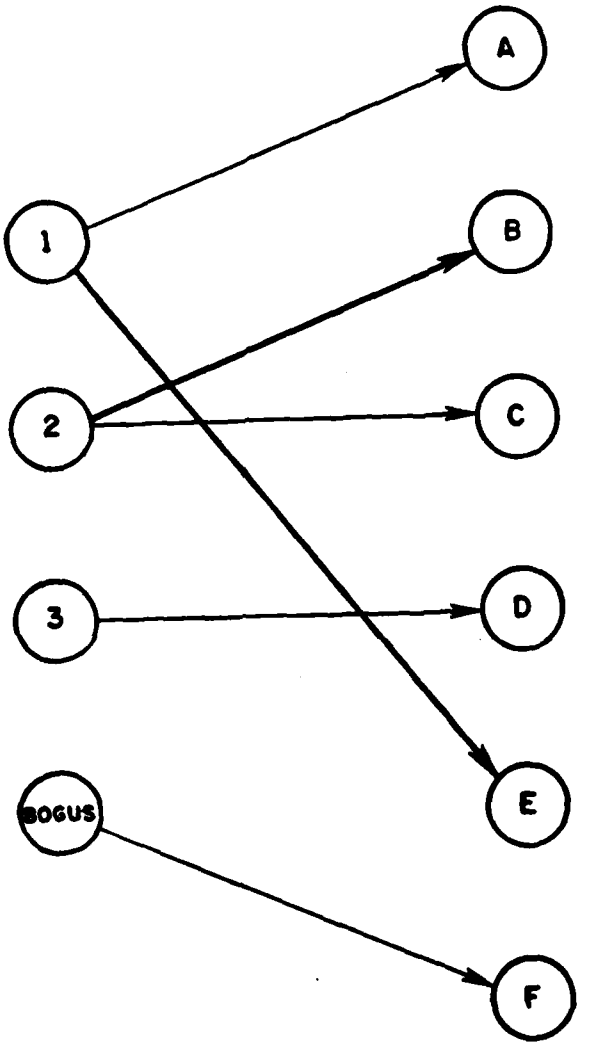


FIGURE 2.5.3. First Swap Assignments (Swapped and newly entered arcs are darkened)

Now pilots 2 and 3 are qualified to perform job F. Pilot 2's job C interferes as does pilot 3's job D. Since both pilots would have to drop the same number of jobs in a swap, one compares their B_{ij} . Pilot 3 has a higher B_{ij} thus pilot 3 drops job D which is picked up by the Bogus pilot and pilot 3 picks up job F.

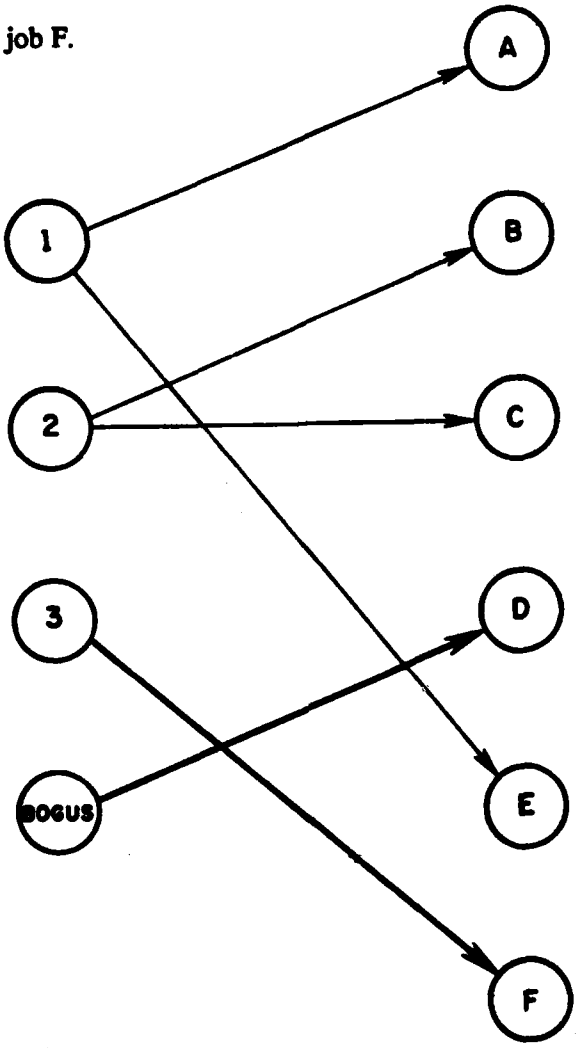


FIGURE 2.5.4. Swap Based on B_{ij}

Pilots 1, 2 and 3 are qualified to perform job D which was just swapped to Bogus. However as job D interferes with job F, (which was "swapped" to pilot 3), pilot 3 cannot pickup job D. Job 4 also interferes with pilot 2's job C and pilot 1's job A. Pilot 1 has a higher B_{ij} and drops job A while picking up job D. A feasible arc exists from pilot 3 to job A thus pilot 3 now picks up job A through a simplex iteration.

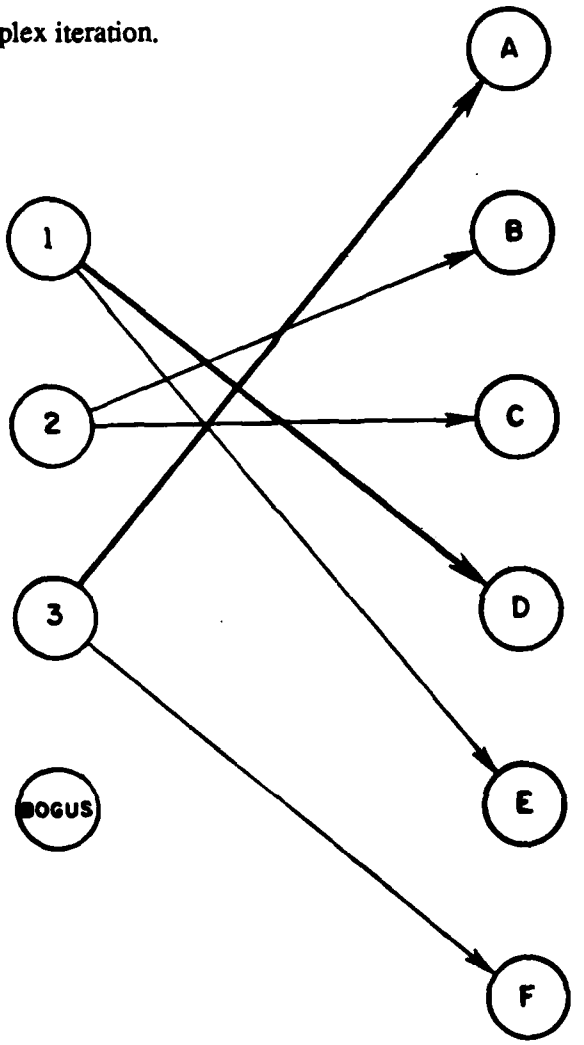


FIGURE 2.5.5. Swap with a Simplex Iteration

If pilot 1 and 2 had not been qualified to perform job D the algorithm would have terminated with job D assigned to Bogus.

The following is a summary of the revised heuristic with swapping and job categorizing.

ALGORITHM C

STEP 0 through 2. Same.

STEP 3. If all category c jobs are assigned go to step 9. Else determine the infeasible arcs and set their cost to M. If any feasible arcs exists for unassigned jobs go to step 1.

STEP 4. Determine I_j = Set of pilots i qualified to perform unassigned job j . Note that if Pilot i is assigned any conflicting job n which was "swapped" to him that he is not "qualified" to perform job j. If I_j is empty assign job j to the "Bogus" pilot and go to step 3.

STEP 5. Determine I_y = Set of pilots x who are available to perform conflicting job $y \in J_j^{'}$ assigned to pilot $i \in I_j$.

STEP 6. Find a pilot $i \in I_j$ for which there are pilots $x \in I_y$ for all conflicting jobs y assigned to pilot i. In case of ties select the pilot with the highest B_{ij}. If found then go to step 8.

STEP 7. Find the pilot $i \in I_j$ with the lowest number of conflicting jobs y for which there are no pilots $x \in I_y$. In case of ties select the pilot i with the highest B_{ij}

STEP 8. Assign job j to pilot i and unassign all conflicting jobs y and place them in category c. Goto step 1.

STEP 9. Print schedule.

In summary this procedure attempts to swap jobs to resolve infeasibilities while disrupting the present schedule as little as possible. Should infeasibilities still exist in the final solution,

schedulers can often resolve them quickly by relaxing constraints such as the length of a particular job. In fact one could program such relaxations however the increase in complexity would likely weigh against the benefits of such a procedure. Resolving such problems interactively has worked well in similar models as shown in [21]. These interactive aspects are discussed in more detail in Chapter 4. Figure 2.6 shows a flow chart for the daily scheduler.

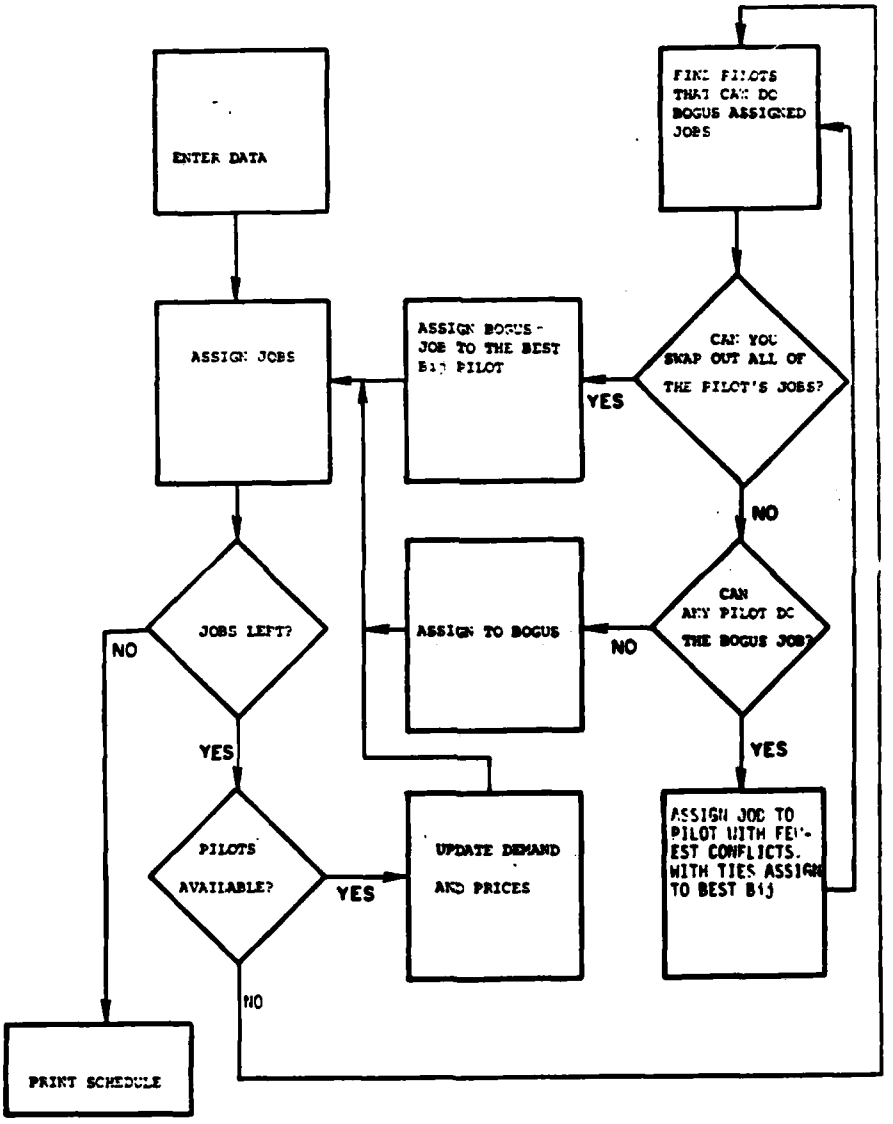


FIGURE 2.6. Daily Scheduler Flow Chart

2.3. Increasing the Time Horizons

Note that one can select any desired time horizon for assigning jobs. However the dynamic environment in which flying sorties occur often degrades planned training into alternate missions or mission aborts. Thus schedules created based on anticipated accomplishment of specific training events occurring at specific times will quickly lead to distorted scheduling. Indeed one of the primary goals in developing this heuristic is to allow schedulers to react quickly to unanticipated changes in a timely manner. However, a long range schedule, of say a week, may provide some benefit, especially in its assignment of non-flying duties which rarely change. In addition, a pilot can gain a general idea of which events he will accomplish the following week and squadron scheduling can identify possible problems such as a shortage of pilots qualified for a specific jobs. By highlighting possible infeasibilities early, squadron supervisors may be able to rearrange the conflicts which create the infeasibilities. Obviously the longer the time horizon the longer the schedule will take to run. However, after completing the next day's schedule, scheduler's could allow the model to run in the evenings after the completion of daily flying and review the results the next day. Thus they would have an updated weekly schedule each day.

2.3.1 Truncation Effects

As outlined in Chapter 1 and will be discussed in greater detail in Chapter 4 most of the training events that are of a required nature have an associated due date or currency. Thus the very real problem exists that in limiting the time horizon that one may induce serious distortions into the daily schedule due to myopia. As mentioned above the high levels of uncertainty would appear to make attempts at scheduling on anticipated states more than one or two days in advance somewhat dubious in their value. Also the level of complexity involved in advancing the scheduling system beyound a deterministic to a stochastic model may make the PC compatibility goal unattainable. However this point has not been pursued in depth and remains an open area of

research for the purposes of the daily scheduler. A look ahead feature is partially captured in the price structure used to determine the relative measure of benefit of a pilot performing a particular job. This price structuring will also be discussed in Chapter 4.

2.4. Incorporating Additional Constraints

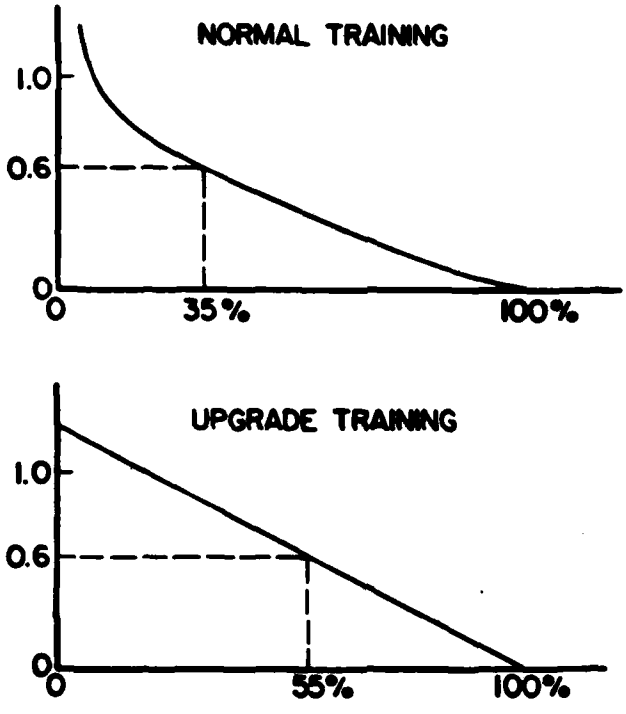
2.4.1. Pairings

As mentioned to earlier, there are advantages to categorization when attempting to pair pilots. A pilot, for example, may have to fly with an instructor pilot to regain flight currency. In addition, tactical fighters, as a matter of doctrine, usually operate in pairs or combinations of pairs with one pilot designated the flight lead and the other a wingman. Some units supervisors like to pair the same pilots for the obvious benefits gained with two individuals always operate as a team. To accomplish this one could divide jobs into flight lead and wingman (or for two place aircraft such as the F-4 into front-seat back-seat). In the manner described in Algorithm C jobs are then assigned by categories. After ground jobs, flight lead positions are filled. Each job a flight lead fills has a paired wingman job associated with it. The paired wingman job has an associated variable indicating which pilot fills the flight lead position. As the price matrix is restructured for the next iteration in step 3 of the algorithm, if Pilot A's paired flight lead occupies the flight lead position then that pilot's price for doing the paired wingman job increases by some factor. One could force a pairing by setting all prices on the arcs to the paired job to $< -M$ unless that individual is the paired wingman for the job. However this increases the likelihood of infeasibilities. A similar logic can work for pilots requiring instructor pilots to regain lapsed currency or to accomplish upgrade training. These pilots are assigned first then a vector indicates the desirability or requirement for an IP in the paired job.

One can prevent pilots from working together in a similar manner. Sometimes supervisors do not wish certain individuals to fly together. After flight lead jobs are assigned, if pilot i is

qualified to perform the paired job of some other pilot x, but pilot i and x have been identified as two individuals who are not to fly together, one simply changes the price on pilot i's arc to the paired job to $< -M$.

As before, setting the value of these pairing factors involves tradeoffs. How much more beneficial is it for pilot A to receive upgrade training with an IP than for pilot B to fly in the same slot on a standard mission? In addition preventing two individuals from working together increases the likelihood of infeasibilities. To give supervisors a clearer picture of the tradeoffs involved in attempting to schedule to satisfy normal training requirements vs pilot pairings, a group of graphs similar to the ones shown in Figure 2.7 may be developed.



a) Select pairing weight.

b) Read right to the intersection off the curve.

c) Read down for the percentage of currency period remaining below which pilot currency will overide a pilot paring in a competing job slot.

FIGURE 2.7. Tradeoffs Between Pairing Pilots and Currencies

For those pilots requiring IPs for currency requalifications the pairing factor should be high as having pilots noncurrent in some event is highly undesirable. For other IP jobs the tradeoffs are less clear. A TFS has a requirement to complete upgrade training (which requires IP's) in a specified period of time but this should not be at the expense of proficiency flying for those pilots who are already combat ready. Obviously empirical experimentation and to an extent intuition will have to be used in determining relative weights.

2.4.2. Length of Tour Constraints

Another constraint which is incorporated into the basic daily scheduler is the crew rest and duty day constraint. Pilots are restricted from having a duty day of greater than L_d hours where the duty day begins at the start of the first job and ends after the last job is completed. In addition, pilots must have L_c hours of uninterrupted free time following their last duty before they can start a new job the next day.

To enforce these constraints one simply appends those jobs to J_j' which would cause duty day or crewrest violations if job j is assigned to pilot i. The length of the duty day and crewrest are user selected though current Air Force regulations specify a 12 hour period for both. So for the standard TFS model all jobs which start less than 12 hours after job j ends and end more than 12 hours after job j begins are appended to J_j' since they would violate the duty day restriction. Likewise all jobs which end less than 12 hours before job j begins and start more than 12 hours before job j begins are appended to J_j' as these jobs interfere with crewrest. In this manner crewrest constraints are enforced for the entire time horizon during any one run of the scheduling program.

After all jobs are assigned one updates the pilots nonavailability due to crewrest constraints in the availability file. The availability file is just that, an input file which tracks a pilot's availability. The data from this file, which also includes manually scheduled jobs as well as long range

scheduled jobs, is used an input into subsequent runs of the daily scheduler. In this manner jobs scheduled in the future are not scheduled in conflict with crewrest constraints from past schedules.

2.4.3. General Constraints

One can enforce constraints of a general nature through the appropriate structuring of the J_j' set. For example by appending all jobs of the same type as job j to J_j' one will exclude an individual from performing more than one type of job j for a given time horizon. In general if by performing job j an individual is excluded from job k, simply append job k to J_j'.

2.5. Improving the Solution - An Interactive System

The Algorithm C provides a solution to the scheduling problem. Again though, one has no guarantee of feasibility much less optimality. Empirical results on a limited number of test problems (see Tables 3.1,3.2 and 3.3) have been encouraging in both the perceived quality of the solution and the execution times. However one may be able to improve the solution while meeting required time criteria. The time available to generate a solution will vary. Preparing Wednesday's schedule Monday evening one may be able to allow the algorithm to run all evening. However on Tuesday morning one may need a schedule within a minute as one react to changes brought about as a cold front moves through. Under such variable conditions allowing the user to select the desired level of optimality, with the attendant time increases for solution, would seem an advantageous way to implement the daily scheduler. Under this scheme Algorithm C is run with all jobs in one category. In most cases this will produce an acceptable schedule in the minimum amount of time. If infeasibilities exist however, the current solution with infeasibilities is presented onscreen while the scheduler program restarts with the two categories mentioned earlier. For relatively minor conflicts the operator may be able to resolve them before a new computed solution appears by simply relaxing constraints. If so he can manu-

ally terminate the program. If not the swapping routine with job categories will most likely resolve any conflicts if possible. Such a feasible solution will likely be "good" enough. If time permits though, the user may elect to let an improvement routine attempt to find a better (higher objective value) solution. This improvement routine is a modification of the swap routine given in section 2.2.

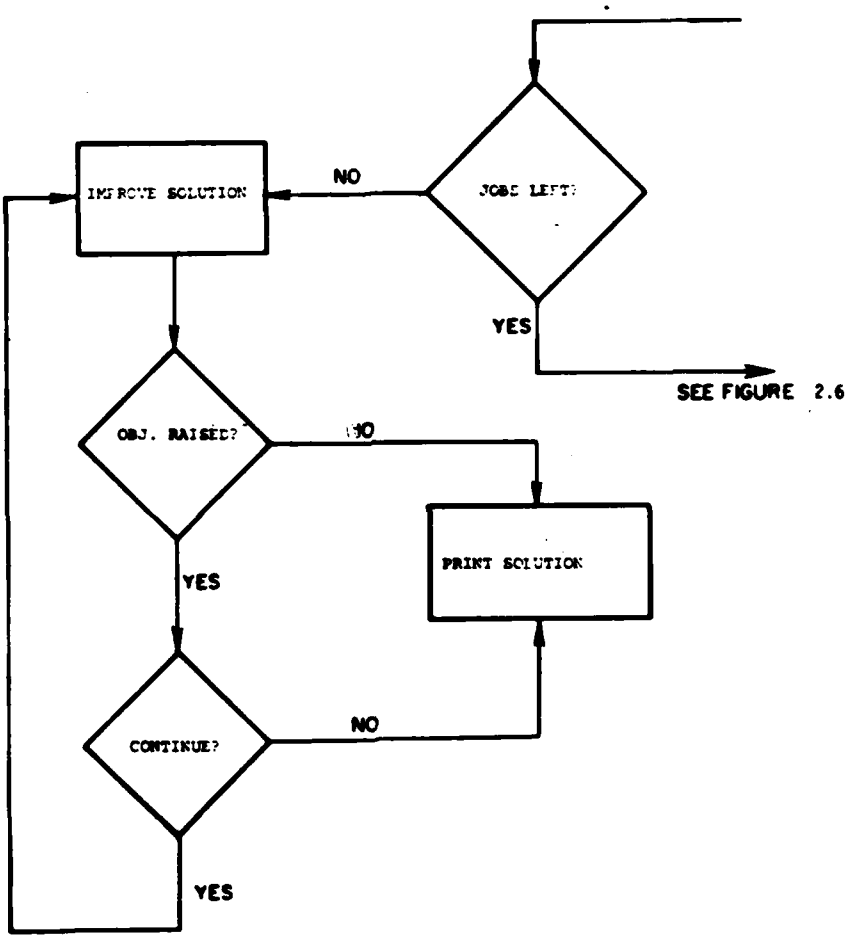


FIGURE 2.8. Appended Flow Chart for Algorithm C With Improvement Routine

In the improvement phase one starts by rank ordering all of the assigned jobs in ascending order based on their arc prices. Starting with the lowest priced job as the input job j one initiates the swap routine. If the pilot who is currently assigned job j can take all of the conflicting jobs y of pilot $i \in I_j$ then one checks to see if such a swap will lower the objective value. If so the swap is made. In case of ties one uses the higher objective value increase to determine which pilot gets the job. This procedure continues until the last job is checked. Jobs are then reordered based on their new (if any) arc prices and the procedure repeats. This recurs until manually terminated or one cycle through all of the jobs can produce no objective value improvements. Note that if we enter the improvement phase with an infeasible solution that a possibility exists for finding one during the improvement phase even though the swap routine failed to do so. This is because one is looking at the jobs in a different sequence and one does not enforce the restriction of not swapping out jobs that have been labeled to a pilot. In addition one is evaluating candidates for swapping based on raising the objective function as opposed to minimizing disruptions. This slight change in logic in effect perturbs the solution that might exit the swap routine and thus may eventually lead one down the road to feasibility.

To summarize the improvement phase:

IMPROVEMENT ALGORITHM

STEP 0: Rank order all jobs 1... m-1 in ascending order by arc price. Set j = 0.

STEP 1: Set j = j+1. If j = m go to step 6.

STEP 2: Determine I_j as before except pilots are not precluded from I_j if they are swapped any job. If I_j empty go to step 1.

STEP 3: Determine I_y. I_y can only contain the pilot who originally had job j . If empty go to step 1.

STEP 4: Find each $i \in I_j$ for which pilot x determined in step 3 can take each y assigned to pilot i. If none to go step 1.

STEP 5: Find the i in step 4 whose swap would cause the maximum increase in the objective value. Make the appropriate swap if the objective value is raised, else go to step 1.

STEP 6: If any swaps have been made go to step 0 else print solution.

Note this is essentially a modified 2-opt improvement procedure [9][39]. In this case however assigning job j to pilot i may mean dropping more than one job from i . Conversely the pilot x that picks up a job dropped by pilot i on a swap does not drop any job other than the job j

CHAPTER 3

RESULTS

3.1 The Network Simplex Code

To solve the network problem a primal network simplex code is used. To start the algorithm one requires an initial solution. Various methods exist to get such a solution but one of the simplest and fastest is the all artificial start or "Big M" method. To see how this method works note that $S_m - D_n$ gives the aggregate excess supply if positive, or aggregate excess demand if negative. Using this number one can merge the Bogus and sink node into one node which is called the root node. Supply arcs connect the root node to the job nodes and demand arcs connect the root node to the pilot nodes. By setting the flow on all of these arcs equal to the corresponding supply or demand from the corresponding pilot or job nodes one gets the initial solution to the transportation problem as depicted in Figure 3.1.

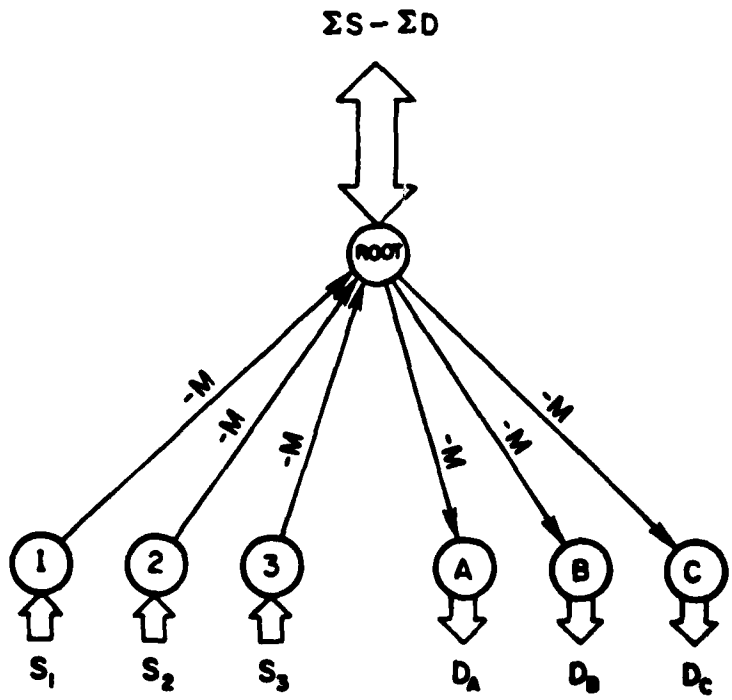


FIGURE 3.1. Initial Solution Using the Big M Method

Before proceeding further a clarification needs to be made about arc prices. The primal network simplex code used here actually seeks to find the minimum cost to the objective function. Consequently, on maximization problems one actually enters negative arc prices on the arcs. In this manner one gets a maximization solution to the problem by just multiplying the results by -1. To be consistent "good" arc prices will continued to be written as posi tive values while negative arcs values indicate "bad" values.

All of the artificial arcs actually have a value of $< -M$ in the code. Recall also that any infeasible arc also has a price of $< -M$. Consequently if any of the arcs in Figure 3.1 are replaced with arcs having prices $\geq -M$ the objective value will improve. The network simplex algorithm conducts this interchange of arcs until no further exchange will improve the solution. To see how this is done note the character of the graph shown in Figure 3.1. All arcs span downward from the root node. This rooted spanning tree graphically displays a possible solution, or basis to the scheduling problem. Now one would wish to see if one could improve it. Consider the possible entry of an arc from pilot 1 to job A. If this arc enters the "basis" the arc from the root to the job A node will have to leave since the arc indicates who is performing the job and only one "pilot" can perform the job (this isn't quite correct but will serve for pedagogical purposes now). Consequently the arc connecting the root to job A is "cut". Thinking of the job A node as a ball on the end of a string one can visualize it falling as the root node arc is cut and the ball swings down to hang under the pilot 1 node (Figure 3.2).

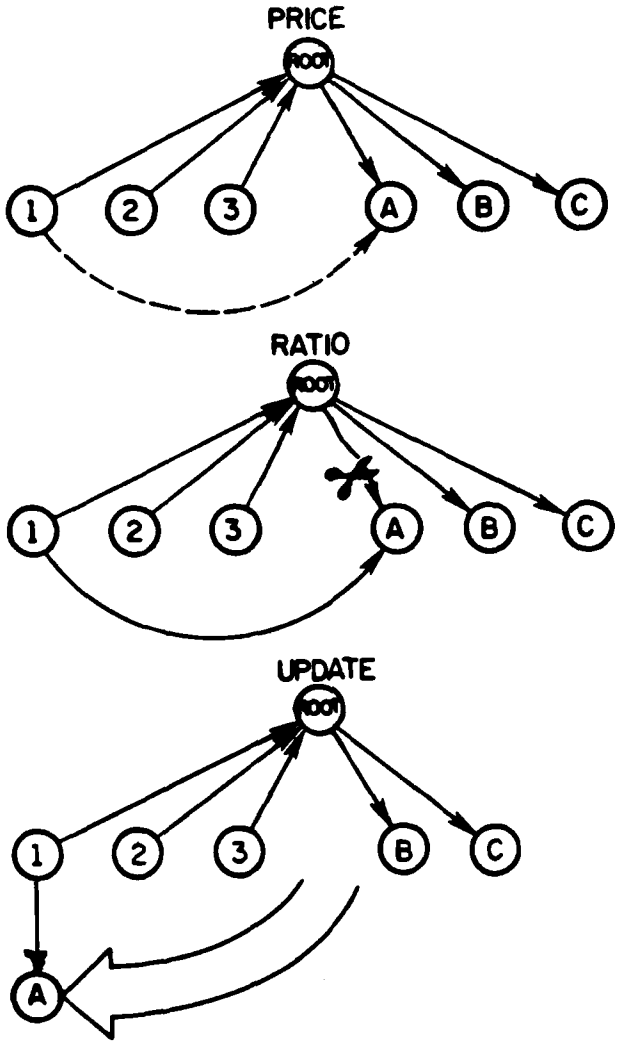


FIGURE 3.2. A Network Simplex Pivot

Finding a node whose entry into the basis will lower the objective value is known as pricing. Determining which node will leave as a result of this arc entering is known as a ratio test. Restructuring the tree following this exchange of arcs is called updating the basis and together with the ratio test is called pivoting.

Various schemes exist to price arcs but they all have one action in common which is determining the reduced cost of a potential arc entering (recall that the network simplex code is actually seeking to minimize cost). Consider following a path down from the root node to each node in the tree. Starting each trip from the root node with a sum of 0, add the price of an arc traversed if moving in the same direction as the arc and subtract the cost of the arc if you traverse the arc opposite the direction it points. When you reach a node the net sum is the node potential or dual price of the node. In the previous example one could evaluate the reduced cost of the arc from pilot 1 to job A as follows:

Reduced cost = Dual of the from node + cost of the arc - Dual of the to node

If this reduced cost is less than 0 then the overall objective value is reduced (raised for a maximization problem) if this arc enters. In the network simplex algorithm used here each arc from a given pilot was priced and the one with the largest (if any) reduced cost was pivoted in. This strategy is known as row most negative or outward node most negative rule. Other more complex methods exist but were not considered due to the small size of this problem in network terms.

The routine of pricing and pivoting continues until no arc prices with a negative reduced cost. At this point the solution is optimal and the algorithm terminates.

The actual primal network simplex code is contained within the computer code for the scheduling algorithm located in Appendix F. The code uses a 6 node- length and 3 arc length list to represent the network basis. The node functions consist of the predecessor node, the predecessor arc, thread, depth, up, and dual functions. Denoting a particular node as i and arc as k, the basis tree was described by the Pred(i) and Predl(i) functions. The Up(i) function indica ted

whether the predecessor arc pointed up or down. The Thread(i) and Depth(i) functions were used in pivoting. The Dual(i) was also maintained as a node function. Data was read in Forward Star form . Forward Star indicates that arcs are listed by the from node in ascending order. By storing the arcs in this fashion one did not have to store the "from" node as all was needed was a pointer to the first arc emanating from a given from node. The to node was stored in the Bnode(k) function while arc prices were stored in the Price(k) function. The basis solution was indicated by the flow over the arcs which was stored in the Flow(k) function. As all upper arc bounds were ∞ and lower bounds 0, they were not explicitly stored. The data storage requirements for this code will vary depending on specific coding techniques. Many tradeoffs can be made between storage limitations and execution speed by storing data explicitly in core. Given the storage limitations of PCs though the author is presently developing a minimal storage version of the algorithm. Designating NP as the number of pilots, NJ as the number of jobs, NA as the number of arcs, and NC the number of job categories used the major storage requirements would consist of the following (A constant cost structure is assumed, pairing designations are not enforced) :

The 6 node length arrays described above * (NP + NJ + 1).

The 3 arc length arrays described above * (NA). Note that infeasible arcs can be designated by changing the sign of their arc prices.

A pointer to the first arc of each pilot and a pointer to the first job of each category, i.e. 1 * (NP + NC + 1).

A pointer for each pilot to the first job he is assigned. Each job has a pointer to the next job assigned to the pilot. The last job assigned to the pilot points back to the pilot. The node number of labeled jobs have a negative value. 1 * (NP + NJ + 1).

The start time and length of each job. 2 * (NJ).

The demand and supply levels for each job and pilot. 1 * (NP + NJ + 1).

The set of jobs which conflict with a given job. One does not have to store this as scanning all the jobs start and stop times would give the conflict information. However as this set is used as a general use con straint enforcer and the conflict comparisons are done so frequently in the code the direct storage of this set is deemed beneficial. (NJ) * (NJ) worst case.

This would give an approximate total storage requirement of 9NP + 10NJ + (NJ*NJ) + 3NA + NC + utility storage. For a typical F-15 squadron this would be approximately 3500 to 4000 data elements for a full daily schedule with a high availibility count for pilots. For more details on network algorithms, their structure, and the functions listed above see [13][20][28][29][52].

3.2 Test Results

The algorithm created was named PAS. PAS was tested over a series of 5 test problems (see Table 3.1). These simplified problems were generated from real world data [22]. Test problem characteristics are listed in Table 3.1. Test problems I and II are "typical" daily flying schedules. Test problem III is a schedule under reduced manning as is common during the holiday season. Test problem IV is a reduced manning and sortie problem typical of a squadron deployed to a remote location. Test problem V is a "hard" schedule with only one feasible solution and a price structure designed to induce the initial assignments away from the feasible solution. It is thus designed to show the reliability of the heuristic "category" solution method and the swap routine. Test problems VI and VIII are reduced schedules with reduced number of pilot qualifications. They were designed to allow a comparison with the integer code ZOOM on other than trivially sized problems. Test problem VII is one such trivially sized problem but nonetheless is instructive along with the other problems in showing the increased running times that one may expect with integer codes as problems get larger.

TABLE 3.1

Test Problem Structure

Network Structure			Integer Formulation	
Problem Number	Number of Nodes	Number of Arcs	Number of Int Var	Number of Constraints
PROBLEM I	64	536	542	3727
PROBLEM II	62	527	519	4116
PROBLEM III	55	491	468	3796
PROBLEM IV	21	71	63	149
PROBLEM V	20	34	28	20
PROBLEM VI	27	101	87	257
PROBLEM VII	19	43	32	18
PROBLEM VIII	31	149	135	495

Table 3.2 shows comparable run times with the integer package ZOOM/XMP [44]. For each model (except problem V) three different data sets were used. B refers to the basic data set generated from real world data. R1 and R2 were two different data sets which employed randomly generated arc prices. This data was generated using the UNIX "srandom" function. Random arc prices were restricted between 0 and 100. All runs were on a Princeton University Vax 11/750 operating under Berkely 4.3 UNIX Operating System. PAS was coded in C and compiled under the UNIX C compiler. ZOOM/XMP is written in Fortran and was compiled under the UNIX Fortran 77 Compiler. ZOOM/XMP incorporates a simplex method with a candidate list pricing strategy to find an initial linear programming (LP) solution to the scheduling problem with the integer constraints relaxed. If the solution is not integer, a heuristic is used in an attempt to find an integer solution. If found and within a user specified tolerance the procedure stops else a branch and bound routine is entered. Details of these procedures are referenced in [44]. For the test problems the objective function value found from the heuristic presented here was entered as an incumbent value. ZOOM/XMP terminated when a better or equal solution was found. Thus ZOOM had three options for an integer solution.

1) An integer LP solution.

2) An integer heuristic solution or

3) A branch and bound solution.

As shown in Table 3.2, test problem VIII required over 1400 CPU/seconds to find a solution. Consequently, larger problems were not tested with ZOOM. Three smaller test problems (test problems V,VI and VII) were also tested with ZOOM. In Table 3.2 the number of pivots, the number of swaps done to achieve feasibility (Feas Swaps), the initial feasible solution (IFS) time and objective value (OBJ(1)), the number of swaps made to improve the solution, the final (FFS) solution time and objective value (OBJ(2)), as well as ZOOM solution times are presented.

TABLE 3.2

Computational Data. All times in CPU seconds.

Prob	Simplex Pivots	Feas Swaps	IFS Time	OBJ(1)	Improve Swaps	FFS Time	OBJ(2)	ZOOM* Soln	ZOOM Time
I B	598	11	4.66	585	12	7.22	722	NA	NA
I R1	424	10	3.67	1984	22	7.61	2473	NA	NA
I R2	464	9	4.03	2094	18	7.73	2404	NA	NA
II B	278	1	1.83	566	7	4.05	656	NA	NA
II R1	234	1	1.83	1849	11	4.10	2008	NA	NA
II R2	289	6	2.50	1910	22	5.94	2276	NA	NA
III B	279	5	2.35	699	7	5.88	763	NA	NA
III R1	346	7	3.95	1702	13	6.40	2033	NA	NA
III R2	267	10	2.53	2145	11	4.85	2394	NA	NA
IV B	54	2	.30	262	2	.43	289	HE	64.13
IVR1**	174	1	.72	608	2	.80	715	HE	82.7
IVR2**	198	0	.85	604	1	.95	648	BB	75.75
V	87	12	.58	12	0	.58	12	LP	.63
VI B	35	0	.18	148	0	.18	148	BB	526.80
VI R1	42	0	.34	738	0	.34	738	BB	284.04
VI R2	36	0	.26	846	0	.26	846	BB	1573.23
VII B	22	0	.04	182	2	.09	192	LP	.85
VII R1	26	0	.09	344	1	.12	360	LP	.93
VII R2	26	0	.14	385	1	.18	395	LP	.98
VIII B	94	4	.61	214	2	.83	223	BB	**
VIIIR1	101	3	.66	986	1	.88	1022	BB	1416.95
VIIIR2	98	1	.54	1032	2	.78	1083	BB	**

* NA - Problem not run
HE - Solution found by ZOOM heuristic
LP - All integer LP solution
BB - Solution found by branch and bound

**Problems IVR1 and IVR2 both had infeasible solutions without categorizing. The times above are for PAS with job categorizing active to enforce feasibility.

Problems VIII B and VIII R2 both terminated after exceeding a 10000 LP iteration limit without exceeding or matching the incumbent value.

The solution times of the scheduling algorithm are highly dependent on the structure of the problem. Problems I - V were intentionally structured "hard" to give a large degree of overlap between jobs. This was to both validate the swap routine and the basic scheduling logic. Swapping though took only a small portion of total run time (see Table 3.3). For the scheduling algorithm the largest portion of computational time was spent in pricing arcs during the network simplex portion of the algorithm and in attempting to find improved solutions after the network simplex algorithm terminated. Pricing operations occur both before and after the swap routine starts. Though arcs with a value $< -M$ were not priced they were scanned to see if their arc prices were $\leq -M$. Pricing after the swap routine is entered could be eliminated all together by using the swap routine to reassign jobs freed from pilots as a result of the availability of feasible arcs. This procedure was not followed in anticipation of the swap routine taking longer to execute than it does and the anticipation that several jobs and arcs would enter into the simplex iteration from the swap routine (while in fact few do).

As mentioned earlier ZOOM was only run on very restricted size problems due to excessive memory requirements and run times that grew exponentially on any problem requiring branch and bound. The ZOOM solution times to problems IV and VI are considerably longer than the scheduling algorithm. Though the ZOOM heuristic found a good solution to problem IV reasonably fast this time is still too long to be practical on a PC. As expected times grow exponentially under conditions where branch and bound was required to get a solution (see problem VI). As both problems V and VII had natural LP solutions the ZOOM times were fast but still longer than the scheduling algorithm. Even so neither of these problems represent realistic schedules in terms of the number of jobs to be scheduled or more importantly the number of integer variables (arcs) that normally exist. Interestingly enough is the inverse relationship between the number of arcs, ZOOM, and the PAS codes. With a large number of arcs the scheduling algorithm is more likely to find a solution without having to resort to many swap iterations to achieve a feasible

solution. However, ZOOM works best under conditions where there are few variables (arcs) and thus few constraints.

3.3 Conclusions

These times show that one can quickly achieve good solutions with the scheduling algorithm. Initial solutions were on average within 9% of the final improved solution in terms of objective values. However this does not indicate how close the final PAS solution was to the true optimal solution. The ZOOM routine did give LP solutions to problems V through VIII. This provided an upper bound as no integer solution can exceed the LP solution. For problems VI and VIII this upper bound was on average 15% more than the final PAS solution found. Again though, this is not necessarily indicative of how close the PAS solution was to the true integer optimal solution. Since the ZOOM code found LP integer solutions to problems V and VII one knows the true optimal value in these two cases. In both of these problems the PAS algorithm also found the optimal solution. However the small size of these problems and the high ratio of pilots to jobs made the initial solutions to these problems exactly the same or very close to the LP solution. This will not in general be the case. In the full size problems I,II, and III the initial PAS solution was 10% to 15% worse than the final PAS solution found. Again one does not know how close the final solution is to the true optimal. Yet how much better the improved solution is over the basic solution much less how much better the "optimal" solution is to the final PAS solution is also a subjective judgement. The improvement routine and "optimal" solutions tend to swap jobs out from low value users and assign them to individuals with high arc prices. A few individuals are assigned all of the jobs. Though it is true that these individuals with high arc prices need to fly more than those with low arc prices, there is a point of diminishing marginal returns. These diminishing returns are not reflected in the formula tions for B_{ij} . A scheduler is likely to prefer a schedule with individuals assigned one job each rather than one where just a handful of people are assigned all of the jobs. To counteract this one could, at the expense of

increased complexity, data storage, and run times, change the B_{ij} between iterations (just as arc prices are changed to $< -M$ due to infeasibility) to reflect job assignments from previous iterations of the PAS algorithm. By incorporating an appropriate penalty term one may price arcs in such a way that an individual is unlikely to be scheduled twice. However it is not clear that such a scheme is desirable either. Some unit schedulers may argue that concentrations of training such as may occur without a penalty term incorporated into B_{ij} are in fact good. This would allow an individual to build on lessons which are fresh in his mind. In addition those who are not scheduled have an entire day free without interruption and thus are able to accomplish their ancillary duties efficiently. As the pricing schemes will get updated the following day anyhow, these individuals will likely fly tomorrow. Thus in the end everyone gets the same number of jobs but assigned in a more efficient manner [22]. This is why an interactive system has been developed. A scheduler is able to generate a range of solutions quickly and then use his judgement to decide what is best. These interactive aspects are discussed further in chapter 4.

TABLE 3.3

Breakdown of Execution Times

Prob	Pivot	Price 1	Price 2	Assign	Change	Swap	Improve
I B	.96	.79	1.41	.26	1.12	.10	3.80
I R1	.65	.86	.88	.22	.93	.10	5.17
I R2	.82	1.11	.98	.24	.77	.09	3.67
II B	.41	.67	.32	.12	.31	0	3.32
II R1	.38	.69	.31	.16	.30	0	3.30
II R2	.43	.58	.51	.19	.71	.07	4.47
III B	.53	.67	.29	.17	.60	.06	4.82
III R1	.66	.96	1.25	.20	.77	.08	3.62
III R2	.46	.76	.53	.19	.47	.11	3.35
IV B	.05	.11	.03	.02	.05	.02	.22
IV R1	.21	.13	.24	.03	.02	.01	.17
IV R2	.10	.59	0	.07	.02	0	.19
V	.09	.10	.11	.04	.03	.03	.01
VI B	.09	.08	0	.01	0	0	.14
VI R1	.07	.11	0	.02	0	0	.14
VI R2	.06	.05	0	.02	0	0	.13
VII B	.03	.01	0	0	0	0	.07
VII R1	.05	.04	0	0	0	0	.06
VII R2	.05	.08	0	.01	0	0	.07
VIII B	.18	.10	.14	.05	.09	.03	.45
VIIIR1	.13	.25	.09	.05	.12	.02	.41
VIIIR2	.18	.14	.08	.05	.08	.01	.45

CHAPTER 4

INCORPORATING GOALS AND LONG RANGE USES

Having concentrated up to now on the basic model formulation and how it might be used to create a daily schedule this chapter links these final results to the beginning inputs. By approaching the problem in this reverse chronological order one can see how the scheduling system will interact with squadron supervisors and schedulers to aid in squadron training management.

4.1. Incorporating Goals

The model is designed to interact with squadron supervisors to produce a schedule which best meets stated squadron training goals. Supervisors form these goals based on direction from higher headquarters (HHQ) and what they perceive as areas to emphasize in training. Goals are first sent down from the next higher level of supervision, normally wing headquarters. The squadron then sets specific goals designed to meet and enhance these wing goals. Finally each functional area in the squadron emphasizes those squadron goals which their particular functional area deals with (see Appendix A).

One can put many of these scheduling goals into the daily scheduler. To do so requires careful structuring of the prices of particular pilots performing specific tasks and a basic understanding of how the scheduling algorithm works.

First , as stated earlier, many tasks are directed by Air Force regulations and manuals. Pilots must accomplish specified levels of events over a 6 month training cycle (see Appendix C). Many of these events are not scheduled but occur during regularly scheduled missions. VID (Visual Identification of an "enemy" aircraft) for example can occur on any mission where meteorological conditions allow. Generally pilots are responsible for accomplishing these events during their missions. Units receive a computerized listing which shows how many and what

events each pilot has remaining to accomplish during each 6 month training cycle. Generally there is no problem in accomplishing these unscheduled events as pilots are daily aware of what events remain and have enough flexibility during normal missions to accomplish them.

Some events can cause problems. For example, airborne refueling requires an airborne tanker aircraft which must be coordinated for and scheduled from an outside organization. Given the difficulty in acquiring such assets, the sorties during which air refueling occurs are generally scheduled based on the need for air refueling training versus the need for training in the sortie type. Sometimes however the sortie requires scheduling priority. Selecting one or the other a priori is difficult since the relative importance of the event versus the sortie depends not only on how many events or sortie types are required to fulfill remaining requirements but how many of the sortie types will be scheduled in the future. In the program developed here the event and the sortie during which it occurs are combined to form a new and unique job. A pilot must be qualified to perform both the sortie and the event to qualify for the new job. His price of performing the job becomes the combined weighted price of performing the job and the event. How the event and sortie are weighted is discussed later.

Another factor affecting price is the currency requirements. For example, pilots are required to fly at least one air combat training sortie every 30 days. If not, they are considered noncurrent and thus nonproficient in air combat training. If a pilot is noncurrent he must fly with an instructor pilot, a limited resource, to regain his currency. Thus a scheduling goal is to not only ensure a pilot accomplishes the specific number of events but maintains his currency.

In addition one must also consider the actual number of events accomplished. As shown later a distinction is further made between the number of events accomplished below the levels directed by regulations and the total number of events accomplished above these levels.

Finally, in addition to the regulation directed requirements, there are usually subjective requirements imposed by squadron supervisors. These too can be structured as mandatory

training requirements, though care must be taken in doing so that actual HHQ directed mandatory training is not lost. For example a commander must ensure that in upgrading all new pilots to combat ready status in less than a squadron mandated 45 days that he does not adversely affect the combat readiness status of his current pilots by preventing them from meeting currencies on their proficiency training.

4.1.1. Setting Prices

The following is an example of how a commander may input his goals to the scheduling system. Say event A must be accomplished 6 separate times over the 6 month cycle. One could represent the price of pilot i doing this event as

$$B_{ij} = \%\text{of events or sorties remaining}$$

for the 6 month training cycle.

One can further modify this price by adding in a training period factor. For example

$$B_{ij} = \frac{\%\text{ events or sorties remaining}}{\%\text{ of training period remaining}}$$

Call the above term the Pro Rata Factor (PRF) as it represents a prorata training accomplishment measure. For example, if one has $\frac{1}{3}$ of the 6 month training cycle left, with $\frac{1}{3}$ of the events remaining this gives a B_{ij} of 1. With $\frac{1}{3}$ of the training cycle and $\frac{2}{3}$ of the events remaining one has a B_{ij} of 2. With $\frac{1}{6}$ of the events remaining one has $B_{ij} = .5$. Thus B_{ij} represents deviation from prorata training accomplishment as a deviation $\pm$ from 1.0. Once all required events for the training period are accomplished, PRF is set to zero, as there is no training left to be met in the training cycle.

One may think of the 6 month training cycle as a currency period in which events must be completed by 30 Jun or 31 Dec. In addition some events have currencies established by higher headquarters (HHQ) which may be shorter or longer than the 6 month cycle. Furthermore, squa-

dron supervisors can impose their own currencies more restrictive than those of HHQ. Since all currencies are not the same and some events are more difficult to accomplish over a short time frame one can modify B_{ij} to reflect currency weighting by adding the following currency factor.

$$\text{Currency Factor (CF)} = \frac{1}{\text{\% of Currency Period Remaining}}$$

Thus B_{ij} becomes

$$B_{ij} = \text{PRF} + \text{CF}$$

Note the currency period is updated each time an event is accomplished. If there is no specified currency period the currency period is set equal to the training period and remains equal to the percentage of the training period remaining throughout the training cycle. As the above denominator term goes to zero B_{ij} is set to M (recall B_{ij} is a price thus it is actually -M in the code). This will ensure a pilot is scheduled as soon as possible if he becomes noncurrent in an event.

One should also include a factor which captures how recently a pilot flew. Note this is somewhat different than percentage of currency period remaining in that it is a measure of how long it has been since a pilot flew rather than how long he has remaining to accomplish a specific event. For example, assume Pilots 1 and 2 both must accomplish an event with no currency specified but with a training period set as a maximum of 45 days. Pilot 1 last flew 5 days ago and Pilot 2 last flew 10 days ago. Both pilots have accomplished 50% of training required and have 44% (20 days) of their training period remaining. Under the original formulation

$$B_{ij} = \text{PRF} + \text{CF}$$

which gives us

$$B_{ij} = .50/.44 + 1/.44 \sim 3.3$$

for both pilots 1 and 2. However most manual schedulers would fly pilot 2 first since he has not flown for the longer period of time. Thus a recency factor (RF) is added where

$$RF = \frac{\text{\# events req} \bullet \text{\# days since last flown}}{\text{\# days in training period}}$$

Finally one should also address the question of total event accomplishment beyond the number required during the training period. Thus a third factor is added to reflect total event accomplishment

$$\text{Event Factor (EF)} = \frac{Z_j}{\text{\# of events accomplished} - \text{\# of events required}}$$

where Z_j is a scaling factor for job j. For the F-15 TFS a $Z_j = \frac{100}{\text{\# of events required}}$ is used as the highest number of events required for any event is approximately 100. Thus Z_j represents a normative weighting for a particular event j. If the number of events accomplished $\leq$ events required $EF = 0$. This gives us a final B_{ij} of

$$B_{ij} = PRF + CF + RF + EF$$

where

$$PRF = \begin{cases} \frac{\text{\% events Rem}}{\text{\% training period rem}} & \text{, if EF} \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$CF = \frac{1}{\text{\% currency period rem}}$$

$$RF = \frac{\text{\# events req} \bullet \text{\# days since last flown}}{\text{\# days in training period}}$$

$$EF = \begin{cases} \frac{100}{\text{\# events req}} \bullet \frac{1}{\text{\# events accomplished} - \text{\# events req}} & \text{, if \# events acc} > \text{\#events req'd} \\ 0 & \text{otherwise} \end{cases}$$

4.2. Determining Sortie Types

As mentioned in the introduction squadrons have relatively little control over number of flying hours and sorties flown. However, they can affect what type of sorties are flown. Here the flexibility of the transportation model proves useful. On average a pilot performs 7-8 scheduled jobs a week of which 3-5 are flying sorties [22]. In aggregate squadrons typically fly around 24 sorties a day. Using this information one can quickly produce a list of what flying jobs would best benefit squadron needs for the coming week. One starts by entering all jobs with a known demand level. For example certain jobs may be scheduled to occur weeks or months in advance. Thus their actual demand levels are known weeks or months in advance. Using the above information one may formulate the sortie prediction problem into a transhipment network as follows.

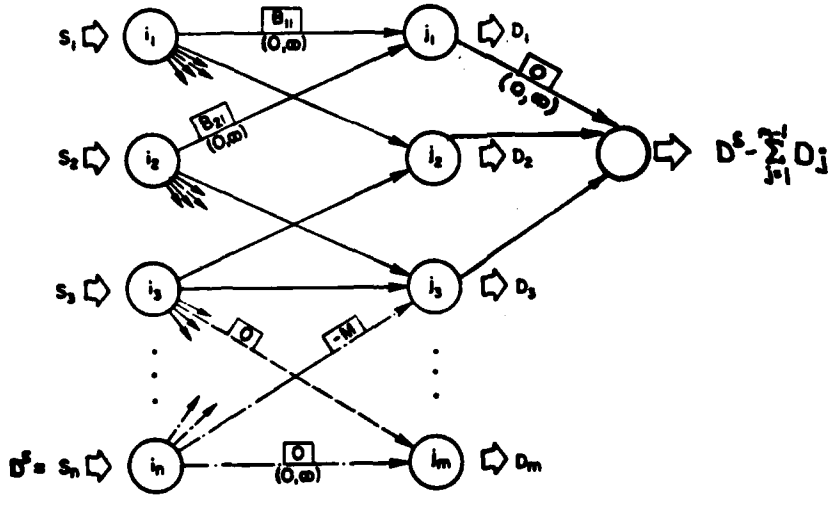


FIGURE 4.1. Long Range Sortie Projection: A Transhipment Problem

A link connects all actual job nodes to a "daily sink". The daily sink demand level is set to $D^s - \sum D$ of all jobs, where D^s is the average number of flying sorties a day (eg 24). The demand levels for each job are 0 unless a specific demand level is known for a given job on a given day. The original sink remains to absorb excess pilot supply (in this formulation supply always exceeds demand). One solves this transhipment problem and gets a prediction of the "best" D^s sorties for any given day. Note that only one transhipment problem need be solved per day as there is no swapping or improvement routine. A similar procedure could be used for ground jobs. After each daily run pilot prices are updated based on the sortie predictions. Then the next day's prediction can be run as described above. Thus a week's aggregate sortie prediction entails the solution of 5 small transhipment problems. One may continue this procedure as long as desired. As pilot qualifications and availability changes, however, these projections become less accurate. A reasonable period of time is two weeks based on the author's practical experience. As mentioned previously the scheduling environment in the TFS is quite dynamic. Even for aggregate sortie projections, predictions beyond two weeks tend to be so fraught with error as to have little use. Armed with this aggregate listing of future job requirements squadron schedules could then request an intelligent sortie mix from the wing schedulers.

The wing (the next level of supervision above the squadron) takes these sortie requests and assigns aircraft specific areas to operate in. In addition they dictate aircraft takeoff times to coincide with the times the airborne operating areas are to be used. Wing scheduling also coordinates outside assets which are used in the support of local training. After the wing scheduling shell (a spreadsheet listing operating areas, mission (job) types, and takeoff times) is produced it is fed back down to the squadrons who then assign pilots to this scheduling shell as described in Section 2. Thus one has moved beyond the daily scheduler to an aid which helps determine schedules weeks in advance. Note however that scheduling the system has gained more generality as the time horizon expands. One is not scheduling specific pilots against specific jobs but only

predicting in aggregate numbers what types of flying sorties would be best for the squadron.

4.3. Creating Job Types

By again increasing the generality one can extend the time horizon of the model even further, to not only predict how many of what type jobs one needs to schedule, but to predict what type of jobs one needs to create. To understand this concept recall that in the daily scheduling model pilot qualifications and availability are input to schedule pilots into specific job demands. Availability is then updated and the process repeated until all jobs are filled.

Similarly, prior to the start of a 6 month training cycle, one can input long range availability of pilots. Inputs affecting this include projected arrivals and departures, projected leaves (vacations), and projected TDYs (temporary duty away from the home station -- nonflying). From this one can create an availability roster. In fact, this is done within the scheduling program on a daily basis. As pilots are assigned jobs they are marked non-available during the time periods the job occurs. In addition manual inputs can be made to the availability roster such as when a pilot cannot fly due to illness or a pilot's request for personal time off. Current pilot qualifications of pilots should also be entered. This is easily available through a "letter of Xs". A letter of Xs is literally that, a table with a big X in rows marked by a pilot's name under a column whose heading is a job that pilot is qualified to perform. This listing is maintained in all TFSs. (see Appendix D)

In addition supervisors normally specify the ratio of specialized qualifications they wish to maintain in their squadrons. For example, a commander may wish for 50% of all pilots to be flight lead qualified, 25% instructor pilot (IP) qualified, etc. Based on current qualifications and such desired manning ratios, the supervisor is then presented with an overage or underage of certain qualifications in future months based on projected personnel changes. An underage becomes a demand for a new job. For example, if due to departures, a squadron is going to be short two IPs in October a demand is created for two IP upgrade programs with a completion date of

October. Previous input from the Supervisor tells the long range scheduler that an IP upgrade should take no longer than, for example, 60 days Therefore the long range scheduler creates a demand for two IP upgrade "jobs" starting 1 August. This information is returned to the supervisor who then decides who will enter upgrade training on those dates (though one could program the model to assign pilots to such jobs, much subjective evaluation goes into selecting such pilots thus actual assignments are better left for the supervisor to determine). Since upgrade training draws away sorties from daily training (concentrating them in the IPs who conduct upgrade training) it is desirable to "smooth" out upgrade training and avoid concentrations. These concentrations occur naturally as old pilots leave and new pilots arrive on a yearly cycle with especially large concentrations every three years (these cycles are due to personnel rotating policies). Thus one needs a long enough time horizon to foresee these "humps" and a programmed logic to spread projected upgrade training throughout the year. A one year time horizon should prove adequate for these purposes. With these new jobs created they are fed to the short range scheduler which produces the squadron job projections sent to the wing. Thus these new jobs show up on the scheduling shell from the wing. Then the daily scheduler assigns pilots to these jobs.

4.4. Other Uses for the Long Range Scheduler

One could also include projected squadron deployments in the long range scheduler. Most TFSs deploy half or all of the squadron 2-4 times a year to another base to participate in military exercises or familiarize themselves with contingency operating areas. Generally participation in these exercises is tracked as any other job, thus the transportation model can be used to recommend assignment to these exercises based on projected availability and past participation (if any). A common problem when only part of a squadron deploys on such exercises is that an inadequate mix of pilots remain to accomplish daily training at the home base. By specifying a particular mix of pilots and their qualifications required for these deployments and for homebase training

(i.e. by specifying what jobs to fill) one can get a fairly good idea who will deploy and who will not.

This again leads to another advantage of placing the model on a PC. Squadrons can deploy their PC with them. Thus training accomplishment and scheduling can be accomplished at the remote site as at the home base. One disadvantage will be the general impracticability of tying in to the mainframe computers located at home bases over telephone lines for retrieval of the data necessary to track currencies and job accomplishments. One could circumvent these problems by performing a database update just prior to deployment. Since most deployments do not last more than 30 days the scheduling irregularities generated at the deployment site due to an inaccurate data base should be minimal. If direct database updates were desired one could tie in by modem to the mainframe computer located at most home bases for a daily update of job accomplishments and currencies. This can be done via the military AUTOVON network (similar to commercial WATS lines) even from overseas locations. Again, however, this system would not be reliable. The AUTOVON system is especially "noisy" and subject to sudden preemption from higher priority calls.

4.5. Interactive Aspects

Many of the interactive aspects of the scheduling and training management system have been previously discussed. In this section the interactive role of the human in the loop is enhanced to both show the advantages and disadvantages of the scheduling system.

The first interactive role involves establishing the price structure which will reflect the training goals of the squadron. Again careful considerations must be given to ways in which price structures will interact with each other in bringing about scheduling tradeoffs. As shown before (see section 2.4.1) one can develop graphs to aid supervisors in the selection of the appropriate parameters. However such graphs will have a limited benefit. One can only show the interaction of a limited number of parameters while many exist. Studies have shown that individuals are

usually saturated with more than 5-7 decision variables to decide among. In addition one risks so engrossing the model in complexity that the users cannot understand it and subsequently are reluctant to use it. The price structure described in Section 4.1.1 is a compromise between too much versus too little complexity. Asymmetries will likely occur. To allow the user to "fine tune" his system one may adjust the price structure by inputting user selected parameters. However in the final analysis one is likely to discover a moving target, one which, regardless of the final price structuring used, faces the very real potential of producing a solution in which a human can intervene and produce superior results in just a few seconds. This model does not seek to eliminate such possibilities. The human must be involved to make it work. This interactive role extends beyond simply defining price structures and reliance on regulation directed event levels.

Referring again to the long range uses of the system, supervisors must ensure that their desired manning ratios and personnel mix on deployments actually reflect realistic goals. This in turn influences the determination of certain job creations. Once jobs are created they should be filled or the situation which led to their creation will be left unresolved. This in turn may create future training and scheduling situations which cannot be met by available resources. In short, the users must understand the system, the logic of its results, and once understanding this logic, implement the results or change the logic. This logic will not be static. Goals and personal taste will change both with time and new leadership. Thus this model is structured with the anticipation of continuous review and a relative ease to change parameters to meet these changes and respond to inadequate results.

The second area of interaction occurs with data entry. Obviously incorrect or inaccurate data produces poor results. This system has been produced with available data systems in mind to minimize additional data requirements above those that presently exist (which is not to say that present data tracking is either adequate or excessive). Inputs are necessary for both long and short range uses. None of these inputs are beyond present day requirements. The one area of

change is the requirement for some inputs to be entered into a computer versus an entry onto a piece of paper.

With the data and squadron level supervisor inputs the long range scheduler produces an aggregate sortie prediction for some specified time frame. This listing will predict the seven or eight jobs for a pilot (in the given example) for the coming week. However this prediction is not fixed. Flight commanders (the first level of supervision within the squadron) should be queried on a weekly basis about any inputs they have for schedulers. Flight commanders in turn then ask, or should ask, their subordinates about what areas of training they wish to emphasize. In fact such a system is currently in force in most TFSs as schedulers seek flight commander inputs weekly to aid in the scheduling of their flight assigned personnel [22]. With flight commander inputs the aggregate sortie prediction is further refined by the schedulers to reflect real world scheduling limitations and opportunities such as the availability of outside assets to support training. After squadron supervisor approval, this aggregate sortie prediction is then forwarded to the wing, where, as described earlier, a scheduling shell is created which is then passed back to the squadrons.

With the shell and an updated list of pilot availability and job qualifications the scheduler is ready to assign individuals to jobs. First, manual scheduling inputs are made. One does not allow the wing commander to be scheduled by the, though logical, apparently random process of an optimizer. Such individuals generally tell schedulers where and when they will fly. Other reasons exist for making such manual inputs. For example, supervisors may insist that two individuals fly together while the pricing process described earlier only increased the possibility of two individuals flying together. Such scheduling inputs can be made by fixing variables at a certain level. Essentially this entails making these pilots unavailable during the time the job they are assigned occurs, the elimination of the assigned job from the job list, and an appropriate modification of the pilot's B_{ij} for that particular job type. Thus, while the job assignment prints

out with the other jobs in the solution report it never actually is "assigned" by the algorithm. With fixed inputs the remaining schedule is run much as described earlier under the daily scheduler algorithm.

When an initial feasible solution is found or no more feasible arcs for remaining jobs exist the solution prints out. In most cases such a solution will be acceptable or, if infeasible, easily modified by manually relaxing constraints to remove infeasibilities. If so the scheduler terminates the algorithm which has either proceeded on into the swap routine, (if the initial solution is infeasible) , or into the improvement routine. With each improved feasible solution the solution is presented. Again the scheduler can terminate the algorithm at any time or continue on until the algorithm terminates under its own logic as described earlier.

These results are then passed on to the squadron supervisor for final approval. Should the scheduler or squadron supervisor wish to make changes they can do so manually or with the algorithm. One can swap a pilot out of or into a job with the swap routine. If possible the swap routine runs until a feasible solution is found or termination criteria is met. With the final results the solution is posted and the availability file updated in anticipation of the next days scheduling input.

As the day goes on scheduling changes may occur. Again the algorithm allows locking in variables which have not changed and reoptimizing the rest or conducting individual swaps. The procedure should be fast enough where schedulers can produce complete daily schedules quickly on a PC computer.

Again note the high level of user interface. This not only allows greater flexibility but the ability to avoid greater complexity in the model structure while still providing good results in a short period of time.

4.6. Scheduling System Summary and Uses

To summarize the planning and scheduling system:

1. Inputs are gathered prior to start of the 6 month training cycle but include personnel projections up to one year in the future. These inputs should be updated monthly thus once initiated the long range scheduler acts constantly. Inputs include:
    A. Projected departure and arrival of pilots
    B. Projected TDYs
    C. Projected leaves
    D. Desired manning ratios
    E. Contingency tasking manning requirements
    F. Squadron established currencies
    G. Squadron established length of upgrade training
    H. Squadron specified sortie mix requirements
    I. Job weights and scaling in accordance with stated goals
    J. Dates of projected deployments and manning structure for deployments

2. With these inputs the long range scheduler returns:
    A. When pilots should enter upgrade training and how many
    B. Projected manning ratios and deviation from goals
    C. Joint leave, TDY, and recommended deployment schedule
    D. Contingency manning levels incorporating projected upgrades

3. Squadron supervisors review this projection and:
    A. Select pilots for upgrade training
    B. Alter TDY leave schedules to avoid problems with required contingency manning

C. Make tentative selection of pilots for deployments

D. Establish pilot pairings if desired

4. This information is fed to the short range scheduler which projects aggregate sortie mix one week to a month in advance.

5. This information is relayed to the wing. The wing combines the squadron's inputs and schedules range air space, and mission types producing a scheduling shell.

6. This shell is passed back to the squadron where the schedulers

A. Manually schedule pilots with nonprogrammed specific requirements

B. Update nonavailability request from individual pilots

7. With this information the daily scheduler

A. Inputs the daily update of pilot qualifications, pilot availability, and the jobs to fill from the wing shell

B. Writes a daily schedule

8. All levels feed back information required to update long, short, and daily schedulers input files.

SUMMARY AND CONCLUSIONS

The advancements in modern aircraft technology have dramatically increased the capability of these aircraft as well as the degree and complexity of the training required for the crewmembers who fly them. During the last 20 years most tactical aircraft have also changed from two seat (two crewmembers) to single seat (one crewmember) aircraft. These factors have combined to create a tremendous burden for crewmembers to accomplish and for squadrons, with their restricted manning, to manually schedule. This paper has outlined a training management system which appears promising in aiding Tactical Fighter Squadrons in the scheduling and management of required daily training.

By formulating the scheduling problem as a transportation network, one is able to take advantage of the speed and efficiency of a primal network simplex algorithm in assigning pilots to jobs. The nature of the daily scheduling problem generally allows the presentation of good results very quickly. However the procedure has the robustness to resolve initial infeasibilities and provide the user with increasing objective value alternative solutions. This procedure continues until the algorithm reaches its own termination criteria, the user stops the problem because of the acceptability of the current solution, or time constraints are met.

A key feature of the scheduling system is the ability of unit schedulers and supervisors to interact at several levels of the scheduling process. Given the complexity of scheduling training it is doubtful that all appropriate constraints can be modeled. Thus this interaction is a necessity to ensure that scheduling and training goals as well as a degree of flexibility is reflected in the scheduling process.

The training management and scheduling system described offers the potential for improved training in a typical TFS. Further research is needed in determining a good measure of how close the objective function is to an optimal value and then guiding the algorithm towards

the optimal within reasonable run times. Along this vein, more advanced improvement procedures such as K-opt methods may prove promising in providing higher objective value solutions within reasonable run times [39]. Given the highly degenerate nature of the solutions to the transportation problems a primal-dual network algorithm may be more efficient than the primal method employed here [40]. In addition, more research is required into integer based algorithms which give quick suboptimal solutions to see if the assumptions are true that such codes would be too slow and too complex to run on a PC. Also other heuristics exist which may work well for this particular problem [7][27]. Finally this model is based on deterministic inputs. The possibility exist for significant distortions in the daily schedules due to a lack of adequate accounting of future requirements though the arc price structures do to a degree take into account future effects. However more research can be done to investigate the possibilty of trying to develop a stochastic system without unduly complicating the scheduling algorithm or increasing its run times beyond the feasible threshold for a PC.

Actual implementation of this heuristic on a PC in a production environment requires further enhancements in software to provide a greater degree of user friendliness and efficiency. In addition interfacing between the data base on USAF mainframe computers and squadron PCs requires further development. Actual implementation also requires more face to face interaction between the developers of the model, squadron scheduler's, and supervisors to resolve conflicts and inaccuracies.

To date results have been encouraging both in the quality of the solution and the speed in which it is found. As of the writing of this paper there are commercial contractors who have approached the Air Force to develop a scheduling system though lack of published material and proprietary concerns prevents a direct comparison. Yet this is indicative of the interest and the probability that someday soon USAF squadrons will operate under some form of computer assisted scheduling and training management.

REFERENCES

1. Baker, E., Bodin, L., Finnegan, W., and Ponder, R., "Efficient Heuristic Solutions to an Airline Crew Scheduling Problem", *AIIE Transactions*, Vol. 11, (1979), pp. 79-85.

2. Balachandran,V. "An Integer Generalized Transportation Model for Optimal Job Assignment in Computer Networks", *Operations Research*, Vol.24, (1976), pp 742-759.

3. Balas, E., and Ho, A.,"Set Covering Algorithms Using Cutting Planes, Heuristics, and Subgradient Optimization: A Computational Study", *Mathematical Programming Study*, 12, (1980), pp. 37-60.

4. Balas, E. and M.W. Padberg, "Set Partitioning" in *Combinational Programming: Methods and Applications*, B.V.Roy (ed.), D. Reidel Publishing Co., 1975.

5. Balas, E., and Padberg, M., "Set Partitioning: A Survey", *SIAM Review*, Vol. 18, (1976), pp. 710-760.

6. Ball, M., Bodin, L., and Dial, R., "A Matching Based Heuristic for Scheduling Mass Transit Crews and Vehicles", *Transportation Science*, Vol. 17, (1983), pp. 4-31.

7. Ball, M., and Roberts, A., "A Graph Partitioning Approach to Airline Crew Scheduling", *Transportation Science*, Vol. 19, (1985), pp. 107-126.

8. Bloomfield, S.D., and McSharry, M.M., "Preferential Course Scheduling", *Interface*, Vol. 9, (1979), pp. 24-31.

9. Bodin, L., Golden, B., Assad, A., and Ball, M., "Routing and Scheduling of Vehicles and Crews -- The State of the Art", *Computers and Operations Research*, Vol. 10, (1983), pp. 63-210.

10. Bodin, L., and Golden, B., "Classification in Vehicle Routing and Scheduling", *Networks*, Vol. 11, (1981), pp. 97-108.

11 Bradley,S.P., Hax, A.C., and Magnanti, T.L., *Applied Mathematical Programming*, Addison Wesley Publishing Co., 1977.

12. Charnes, A., Glover, F., Karney, D., Klingman, D. and Stutz, J., "Past, Present, and Future Development, Computational Efficiency and Practical Use of Large Scale Transportation and Transshipment Computer Codes",*Computers and Operations Research*, Vol 2, (1975), pp. 71-81.

13 Chvatal, V., *Linear Programming*, W.H.Freeman and Co., 1983.

14. *Computer Scheduling of Public Transport: Urban Passenger Vehicle and Crew Scheduling*, A. Wren (ed), North-Holland Publishing Co., 1981.

15. Cornuejols, G., Fisher, M., and Nemhauser, G., "Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms", *Management Science*, Vol. 23, (1977), pp. 789-810.

16. Dantzig, G., and Fulkerson, D., "Minimizing the Number of Tankers to Meet a Fixed Schedule", *Naval Reserve Logistics Quarterly*, Vol. 1, (1954), pp. 217-222.

17. DeGans, O., "A Computer Timetabling System for Secondary Schools in the Netherlands",

European Journal of Operational Research, Vol. 7, pp. 175-182.

18. Derigs, U., "A Shortest Augmenting Path Method for Solving Minimal Perfect Matching Problems", *Networks*, Vol. 4, (1981), pp. 379-390.

19. DeWerra, D., "An Introduction to Timetabling", *European Journal of Operational Research*, Vol. 9, pp. 151-162.

20. Dial, R., Glover, F., Karney, D., and Klingman, D., "A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees", *Networks*, Vol. 9, (1979), pp. 215-248.

21. Dyer, J. and Mulvey, J., " An Integrated Optimization/Information System for Academic Departmental Planning", *Management Science*, Vol. 22, (1976), pp. 1332-1341.

22. Feest, G. Capt., Noted in Conversation, Langley AFB, Virginia, July 1986.

23. Fisher, M., "The Lagrangian Relaxation Method for Solving Integer Programming Problems", *Management Science*, Vol. 27, (1981).

24. Fisher, M., "Optimal Solution of Scheduling Problems Using Lagrange Multipliers; Part I", *Operations Research*, Vol. 21 (1973), pp. 1114-1127.

25. Fisher, M., "Worst Case Analysis of Heuristic Algorithms", *Management Science*, Vol. 26, (1980), pp. 1-17.

26. Geoffrion, A., "Lagrangian Relaxation for Integer Programming", *Mathematical Programming Study*, Vol. 2, (1974), pp. 82-114.

27. Glassey, R., and Mizrach, M., "A Decision Support System for Assigning Classes to Rooms", *Interfaces*, Vol. 16, (1986), pp. 92-100.

28. Glover, F., Karney, D., Klingman, D., and A. Napier, "A Computational Study on Start Procedures Basis Change Criteria, and Solution Algorithms for Transportation Problems",*Management Science*, Vol.20, (1974), pp. 793-813.

29. Glover, F., Karney, D., and Klingman, D., "Implementation and Computational Study on Start Procedures and Basis Change Criteria for a Primal Network Code", *Networks*, Vol.4, (1974), pp. 191-212.

30. Glover, F., Karney, D. and Klingman, D., "Improved Computer Based Planning Techniques- Part 2", *Interfaces*, Vol. 9, (1979), pp. 12-20.

31. Glover, F., Hultz, J., Klingman, D., and Stutz, J., "Generalized Networks: A Fundamental Computer Based Planning Tool", *Management Science*, Vol. 24, (1978), pp.1209-1219.

32. Glover, F., and Mulvey, J., "Equivalence of the 0-1 Integer Programming Problem to Discrete , Generalized, and Pure Networks", *Operations Research*, Vol.28, Part II, (1980).

33. Golden. B., and Magnanti, T.L., *Network Optimization*, Draft 1984.

34. Held, M. and Karp, R. "The Traveling Salesman Problem and Minimal Spanning Trees", *Operations Research*, Vol.18, (1970), pp. 1138-1162.

35. Holloran, T.J. and Byrn, J., "United Airlines Station Manpower Planning System", *Interfaces*, Vol. 16, (1986), pp. 1-9.

36. Knauer, B., "Solution of a Timetable Problem", *Computers and Operations Research*, Vol. 1, (1974), pp. 363-375.

37. Lenstra, J., and Rinnoy, A., "Complexity of Vehicle Routing and Scheduling Problems", *Networks*, Vol. 11, (1981), pp. 221-227.

38. Lin, S., "Heuristic Programming as an Aid to Network Design", *Networks*, Vol. 5, (1975), pp. 33-44.

39. Lin, S., and Kernighan, B., "An Effective Heuristic Algorithm for the Traveling Salesman Problem", *Operations Research*, Vol. 21, (1973), pp. 498-516.

40. Luenberger. D.G., *Linear and Nonlinear Programming*, Addison- Wesley Publishing Co., 1984.

41. Marsten R., "An Algorithm for Large Set Partitioning Problems", *Management Science*, Vol.20, (1974), pp. 774-787.

42. Marsten, R., Muller, M., and Killian C., "Crew Planning at Flying Tiger: A Successful Application of Integer Programming", *Management Science*, Vol. 25, (1979), pp. 1175-1183.

43. Marsten, R., and Shepardson, F., "Exact Solution of Crew Scheduling Problems Using the Set Partitioning Model: Recent Successful Applications", *Networks*, Vol. 11, (1981), pp. 167-177.

44. Marsten, R., *User's Manual for ZOOM/XMP*, Department of Management Information Systems, University of Arizona, November 1984.

45. Mitra, G. and Welsh, A., "A Computer Based Crew Scheduling System Using a Mathematical Programming Approach", in reference [14].

46. Mulvey, J., "A Classroom/Time Assignment Model", *European Journal of Operational Research*, Vol. 9, (1982), pp. 64-70.

47. Mulvey, J., "Strategies in Modeling: A Personnel Scheduling Example", *Interfaces*, Vol.9, (1979), pp. 66-77.

48. Mulvey, J., "Pivot Strategies for Primal-Simplex Network Codes", *J. ACM*, Vol. 25,(1978), pp. 266-270.

49. Nemhauser, G., Wolsey, I., and Fisher, M., "An Analysis of Approximation for Maximizing Submodular Set Functions", *Mathematical Programming*, 14, (1978), pp. 265-294.

50. Ross, G.T., and Soland, R., "A Branch and Bound Algorithm for Generalized Assignment Problems", *Mathematical Programming Study*, Vol. 8, (1975), pp. 1-103.

51. Parker, M., and Smith, B., "Two Approaches to Computer Crew Scheduling", in reference [14].

52. Powell, W., Lecture Notes: Network Algorithms and Applications, 1986.

53. Ryan, D., and Foster, B., "An Integer Programming Approach to Scheduling", in reference [14].

54. Smith, B., and Wren, A., "VAMPIRES and TASC: Two Successfully Applied Bus Scheduling Programs", in reference [14].

55. Srinvasan V. and Thompson, G., "Benefit Cost Analysis of Coding Techniques for the Primal Transportation Algorithm", *J. ACM*, Vol.20,(1973), pp. 194-213.

56. Tactical Air Command Manual 51-50, *Aircrew Training* ,1986.

57. Tripathy, A., "School Timetabling - A Case in Large Binary Integer Linear Programing", *Management Science*, Vol.30, (1984),pp. 1473-1489.

58. Ward, R., Durant, P., and Hallman, B., "A Problem Decomposition Approach to Scheduling the Drivers and Crews of Mass Transit Systems", in reference 14 .

APPENDIX A

Typical TFS Goals

TFS

1986 GOALS

MISSION

TFW/DO Goal: Improve TFW capability to deploy and fight worldwide

TFS Objectives:

(1) Fly mixed force (F-15/F-16) training during at least one Aggressor visit.

(2) Deploy every Mission Ready/Mission Support squadron pilot and every squadron enlisted member on at least one composite force exercise.

(3) Develop quarterly training scenarios for all Oplan-tasked TFS deployment bases.

(4) Fly the FY86 Flying Hour Program out to zero hours/sorties.

TFW/DO Goal: improve local DACT and training realism.

TFS Objectives:

(1) Fly at least 60% of all squadron ACBT sorties against dissimilar opposition (MQT sorties excluded).

(2) Provide coordinated (DOW, IN, DOT, DOX) air tasking orders for an average of at least one mission (two- or four-ship flight) per local training day.

PEOPLE

TFW/DO Goal: Enhance airmanship.

TFS Objectives:

(1) Average less than 45 days for MR upgrade of newly-assigned RTU graduates.

(2) Review pilots for upgrade to flight lead and instructor pilot as soon as they meet minimum criteria. Select those best qualified.

(3) Schedule one "down" Friday per month for ATWATS, Flight Lead seminars, etc.

TFW/DO Goal: Take care of people, their families, and their environment.

TFS Objectives:

(1) Schedule maximum crew duty day of ten hours concurrent with two-go days, except for sortie surges.

(2) Reward outstanding performance by flying maximum supportable maintenance appreciation sorties (average 3/month).

(3) Encourage PME/Off-Duty education participation by accommodating class schedules to the maximum extent possible.

TFS SCHEDULING 1986 GOALS

1. Fly the FY86 Flying Hour Program to zero hours/sorties.

2. Fly at least 60% of continuation ACBT sorties against dissimilar adversaries.

3. Provide the Training Shop with an average of at least one mission (two or four shop) for air tasking orders per local training day.

4. Maintain our current positive Eagle Elite sortie position so as to continuously be permitted to fly an average of at least 3 Maintenance Appreciation sorties per month to enhance TFS/AMU relations.

5. Strengthen TFS/GCI interface by flying a minimum of one GCI controller per month in a D model on an ACTT/DACT sortie.

6. Complete all Stan/Eval checkrides before the fifth month. Complete all prerequisites prior to the end of the third month.

7. Complete all MQT training within an average of less than 45 days.

8. Manage the annual Flying Hour Program to schedule one "Down Day" on one Friday per month for ground training and commander directed meetings.

9. Maximize the use of the two go schedule, IAW TAC goal, while maintaining a maximum crew duty day of ten hours.

10. Encourage PME/Off-Duty education participation by accommodating class schedules to the maximum extent possible.

11. Efficiently schedule resources to attain GCC level B for 75% of MR pilots.

12. Incorporate squadron small computer resources into the schedule planning process.

APPENDIX B

Scheduling Checklist

DUTY SCHEDULER/OPS SUPERVISOR

The Day Prior:

1. Receive a hand-off briefing from the current duty scheduler concerning the next days schedule and any pertinent considerations to include priorities and scheduling rationale.

2. Contact the Base Weather Forecaster and obtain the forecast for the next flying day.

3. On the basis of this information, determine the need for back-up scheduling plans to include:

    a. WX Category Changes.
    b. Mission Changes.
    c. Range Space Changes.
    d. Configuration Changes.
    e. Possible WX adding to the PM Schedule.

4. If the probability of adverse WX, or other extraneous inputs (i.e., VIP's, or exercise) is high, identify primary back-up pilots and notify these individuals of your plans.

5. Take the schedule and availability sheets home with you. If the need arises, you will be called.

The Day Of:

1. Call WX before you leave home -- you may need a head start.

2. Arrive NLT 15 minutes prior to the first briefing.

3. Call the AMU and confirm the following:

    a. Configurations.
    b. Tub Lines.
    c. Number of Spares (A/A and A/G)
    d. ICT Lines.
    e. V.I.P.'s

4. If the configurations are not right or other difficulties are encountered, note it in the Green Record Book so it can be followed up.

5. Take the previous day's schedule and transfer all the sorties flown to the weekly summary so we have a running goal for the week for each pilot. Then take the schedule and file on Chief of Scheduling desk.

6. Review the Ops Scheduling Board for all Upgrade sorties flown and ensure that Training has updated the Training board. Note any discrepancies and give to Chief of Scheduling.

7. As changes to airspace, takeoff times, etc., occur to future days schedules ensure that the changes are recorded on the grease board and the MATS and the Wing Printed schedule (if it is available). All of these sources must be identical.

8. Standby to work the daily fires that occur (Personnel changes, Airspace changes, and Management decisions (Go/No Go, Pit, ICT, etc.)

   NOTE: Let the Ops Specialists do their jobs unless you are not getting results fast enough. If you need higher level guidance talk to the P Sup.

9. Start working the next days schedule once today's situation is in hand and you have completed the above tasks. The best time to get started is early in the morning prior to outside agencies calling us about their screw ups. Remember the systematic approach will get you to the finished product earlier, and that gathering all the information to build the schedule, prior to just throwing names at the schedule, will get you a quality product sooner. Use the following steps to ensure all scheduling factors have been considered.

A. Make a copy of the next day's schedule from the Wing printed schedule (If it is available -- usually after Monday) or the handwritten MATS if Wing printed is not available.

B. Get a blank copy of the ground activities sheet (lists P Sup, Duty Sch, SOF/RSO, RTO, Simulators, and Meetings). List the requirements for the duties by the applicable time period without listing the names to fill these requirements. Get all the meetings that are printed on the large calendar on the wall, as well as the regularly scheduled meetings, which should already be printed on the schedule board.

C. Get a blank deconfliction sheet and list all the Leave/TDY's (Leave/TDY information is posted next to the schedule boards).

   Review the schedule book for the applicable week for any extra requirements (VIP Flights, Eagle Elites, Higher Hq Flyers, Statics/Flybys).

D. Review the Flight CC scheduling inputs and attempt to incorporate their inputs according to the priority listed.

E. Review Life Support requirements sheet and the Calendar of Training dates. Schedule as many of these requirements as the schedule will allow.

F. Start building the schedule by first listing the hard ground requirements and then fill in the flying schedule by using formed elements when able.

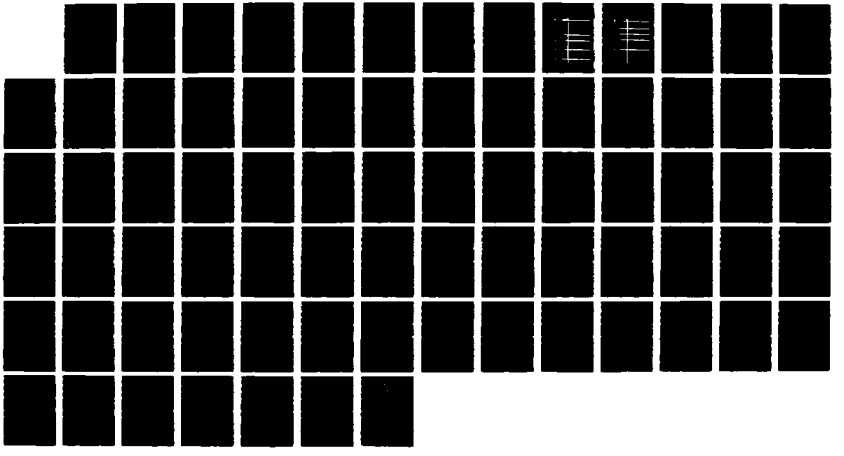
G. Fill in the deconfliction sheet as you build the schedule step by step. This will help to make changes to the work as you change the schedule or someone comes in with a new input. Subsequent supervisor review and the changes which may result can also be handled much

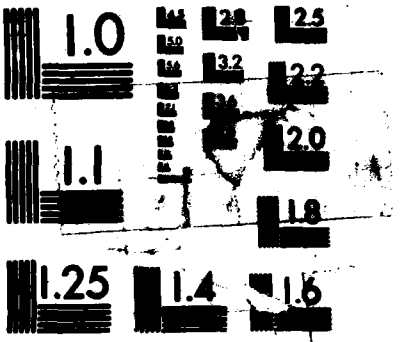
faster. If the deconfliction sheet is not accurate you will screw the next day's duty scheduler who may need to make a quick change to the schedule.

H. If it is Tuesday attend to 0930 Mx/scheduling Meeting held at the DCM Conference Room. Bring the following items to this meeting: The Green Record Book, this week's MATS, and next week's MATS. Be prepared to discuss upcoming taskings for the TFS record any information in the Green Record Book.

I. Attend the daily 1400 meeting to reconfirm the next day's schedule (number of lines, configurations, configuration changes between go's, etc.). These items should be discussed with the AMU schedulers prior to the meeting. Missile availability can be obtained for AMU schedulers or from the AMU directly.

AD-A187 341 A TRAINING MANAGEMENT AND SCHEDULING SYSTEM FOR UNITED 2/2
STATES AIR FORCE T (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH M T MATTHEWS JUN 87
UNCLASSIFIED AFIT/CI/NR-87-89T F/G 5/9 NL





DAILY SCHEDULING CHECKLIST

1. Review the previous days flying schedule and record the number of scheduled/flown sorties.

2. Advise the Operation Clerk to erase yesterday's schedule.

3. After validating the previous days schedule -- use it to do the bean count for who flew, who served RSO, RTO, and SOF.

4. Look at the bean count and see who needs to fly. Evaluate this against other scheduling priorities directed from the TFS/CC, DO, or Flight Commanders. Consider other hard inputs such as checkrides, MQT sorties, and upgrade sorties.

5. Ensure you have coordinated with training so as not to schedule something that needs a prerequisite. This occasionally happens during MQT upgrade when someone had a non-effective sortie.

6. Make sure you have the correct sim, RSO, RTO, and SOF periods posted.

7. If it is Monday or Wednesday, make sure you have scheduled 1400 Egress/Hanging Harness training. If the Sq LSO is scheduled to track that day, make sure he is off the flying schedule that afternoon. Also schedule Egress/Hanging Harness for all VIP/Eagle Elites/Mx Appreciations at 1230.

8. Check the weekly meetings sheet and confirm the next day's meeting are posted. (e.g., 1600 daily stand-up).

9. Check the ground training calendar to make sure that someone doesn't miss scheduled chemical warfare training or .38 caliber pistol training.

10. After all known quantities are up, consider "pop ups". Do we have to schedule a static display pilot? Do we have an aircrew extraction? Have take-off times, ranges, or configurations changed? Did we get "pop up" DACT? Did previously scheduled adversaries cancel?

11. Confirm who is on leave or TDY. Ensure this information is posted on the deconfliction sheet. Also put all meetings, appointments and crew rest requirements on this sheet. Now you are ready to put names on the schedule. USE the deconfliction sheet. Keep it current. Remember, unless you are quick turning someone, you should allow 6 hours between their initial brief time and their next event.

12. Post names for all the hard requirements first, (checkrides, MQT, upgrade sorties, or TFS/DO directed priorities).

13. Post four ship flight leads; then element leads and wingmen in order. Use flight pairings to the max extent possible.

14. Ensure the CC, DO, or ADO are in the squadron during flying operations.

15. Ensure both the CC and DO are not airborne at the same time.

16. If the Wing DO is flying with us make sure the CC is available to take his brick or arrange for some other CC to take it.

17. Run the deconfliction checklist, make sure the deconfliction worksheets are correct.

18. Have your schedule approved by the DO or ADO prior to getting it the Ops clerks by 1200.

19. Make sure Mx gets a copy in time for the 1400 Mx Meeting.

20. Standby for flash floods.

Deconfliction Checklist

-- All lines filled
-- Crew rest first and last go's
-- 4-ship flight leads, 2-ship element leads
-- Aircrew turn times if applicable
-- Ops Sups deconflicted
-- SOF's deconflicted
-- SIM's deconflicted
-- Stand up covered
-- MTG's covered/deconflicted
-- Names deconflicted by Egress, H/H, CW Tng, ITC, etc.
-- Names deconflicted for appointments
-- Remarks reflect all the following where appropriate: DACT phone numbers, TUB, FLUP, IPs, Mass Brief times/places, Area Restrictions, etc.
-- T/O times appropriate for area times.
-- VIP's covered - Egress, escort, etc.

APPENDIX C

TACM 51-50 Excerpts

TFS GUIDE TO 51-50 REQUIREMENTS

JUL - DEC 1986

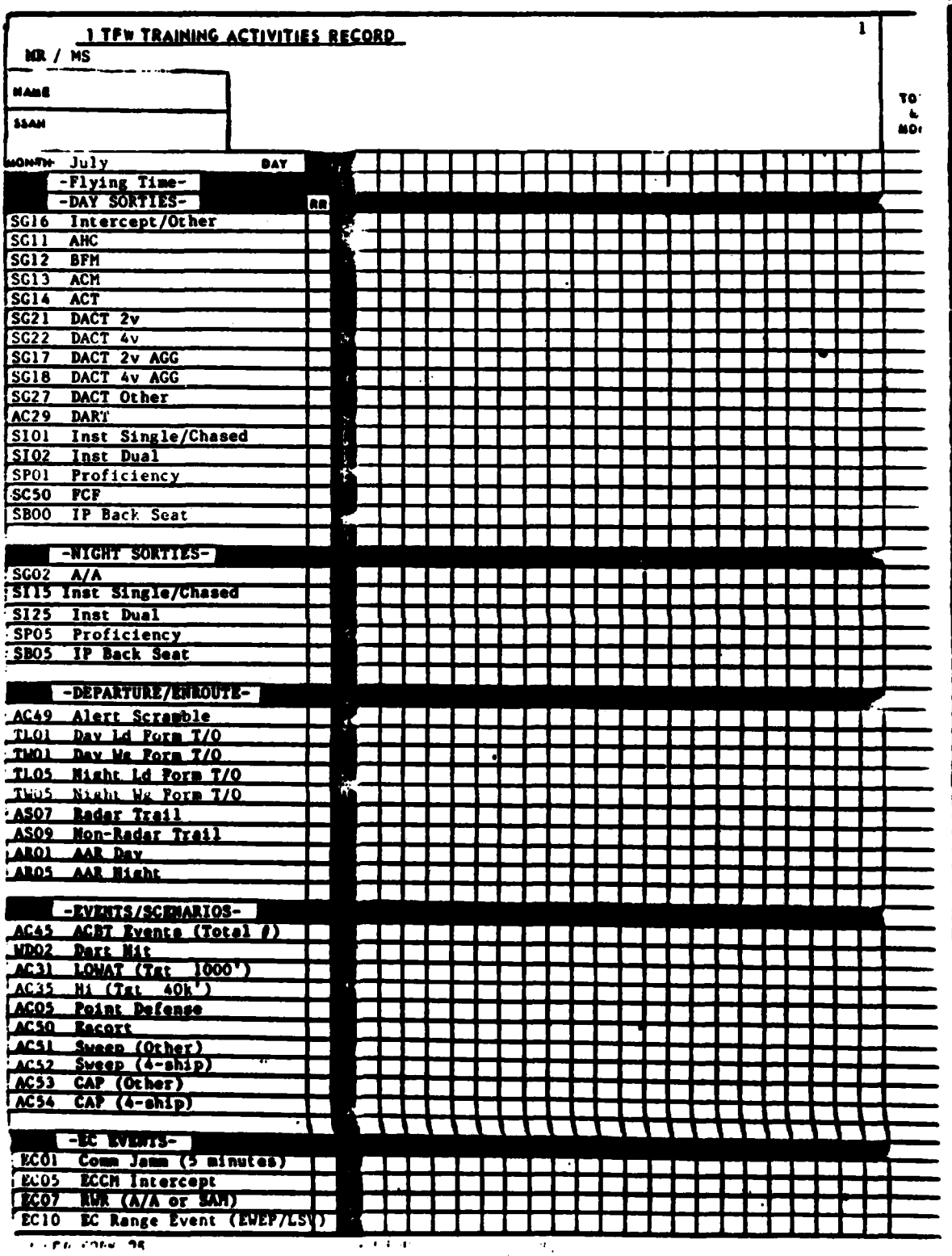
EVENT/SORTIE	MR-EXP	MR-INEXP	MS-EXP	MS-INEXP	CURRENCY	(INEXP/EXP)
TOTAL	A-47 B-59 C-83	A-53 B-70 C-96	30	30		
GCC	A-37 B-48 C-70	A-43 B-58 C-82	5/MONTH	5/MONTH		
AIR/AIR	A-33 B-44 C-66	A-39 B-54 C-78			90 DAYS	
NIGHT A/A	4					
NIGHT SORTIE			2	2		
DART	A-1 B-QUAL C-2	A-1 B-QUAL C-2			12 MONTHS QUALIFY	
CW FLT	1	1			*	
INSTRUMENT	0	2	0	2		
IP FLT	42				60 DAYS	
COMP FORCE SRTY	A-0 B-1 C-2					
RED FLAG	1/15MON	1/15MON				

EVENTS		
ACBT	A-50 B-70 C-98	A-60 B-84 C-119
SWEEP	A-2 B-8 C-12	
4-SHIP SWEEP	A-2 B-4 C-6	
POINT DEFENSE	A-2 B-3 C-6	
ESCORT	A-2 B-4 C-6	
CAP	A-2 B-4 C-6	
4-SHIP CAP	A-2 B-4 C-6	

HIGH INTCPT (ABOVE 40M')	A-4 B-4 C-4				
LOWAT INCPT (BELOW 1M')	A-10 B-14 C-18				60/90 DAY CURRENCY
EC TRAINING					
EC EVENT (RWR)	A-4 B-4 C-8				
EC RANGE EVENT	A-1 B-1 C-1				
COMM JAMM	A-1 B-2 C-3				
ECCM INTCP	A-1 B-2 C-4				
INSTRUMENTS					
PENETRATION	6	6	6	6	INEXP MAY DO UP TO A TOTAL OF THREE APPR IN THE SIM
PREC APPR	12	18	12	18	
NON-PREC APP	12	18	12	18	
HUD OFF PENETRATION	3	3	3	3	
HUD OFF PREC APP	6	9	6	9	
HUD OFF NON-PREC APP	6	9	6	9	
SIMULATORS	6	9			INST & EPs ON EACH: NON GRADED SIM, EXCEPT SIM IPs
SIM IP EP MSN	1	1			
CHEM WARFARE TRNG					
CW FLT	1/HALF		1	1	
CW SIM	1/HALF*		1	1	
AAR	3	3	2	2	6 MONTHS
AAR-NIGHT	1	1	1	1	

FORMATION TAKEOFF					60/90 DAYS
FORMATION LDG					60/90 DAYS
FLT LD WING TO					6 MONTHS
FLT LD WING LDG					6 MONTHS
NIGHT LNDG	2	2	2	2	15/30 DAYS**
IP BACK SEAT LDG					45 DAYS

* TAC GOALS ONLY
` IF A CW EXERCISE FLIGHT IS NOT ACCOMPLISHED, AN EXTRA SIM IS NEEDED.
$ MS AIRCREWS CLEARED TO FLY TACTICAL EVENTS WILL PRACTICE TACTICS IN THE SIMULATOR
$$MS AIRCREWS PARTICIPATING IN THE EC PROGRAM MUST ACCOMPLISH THESE EVENTS
**A DAY OR NIGHT LANDING WILL UPDATE THIS CURRENCY
``FAILURE TO MAINTAIN CURRENCY WILL RESULT IN LOSS OF MR STATUS.



1 TFW TRAINING ACTIVITIES RECORD

1

MR / MS

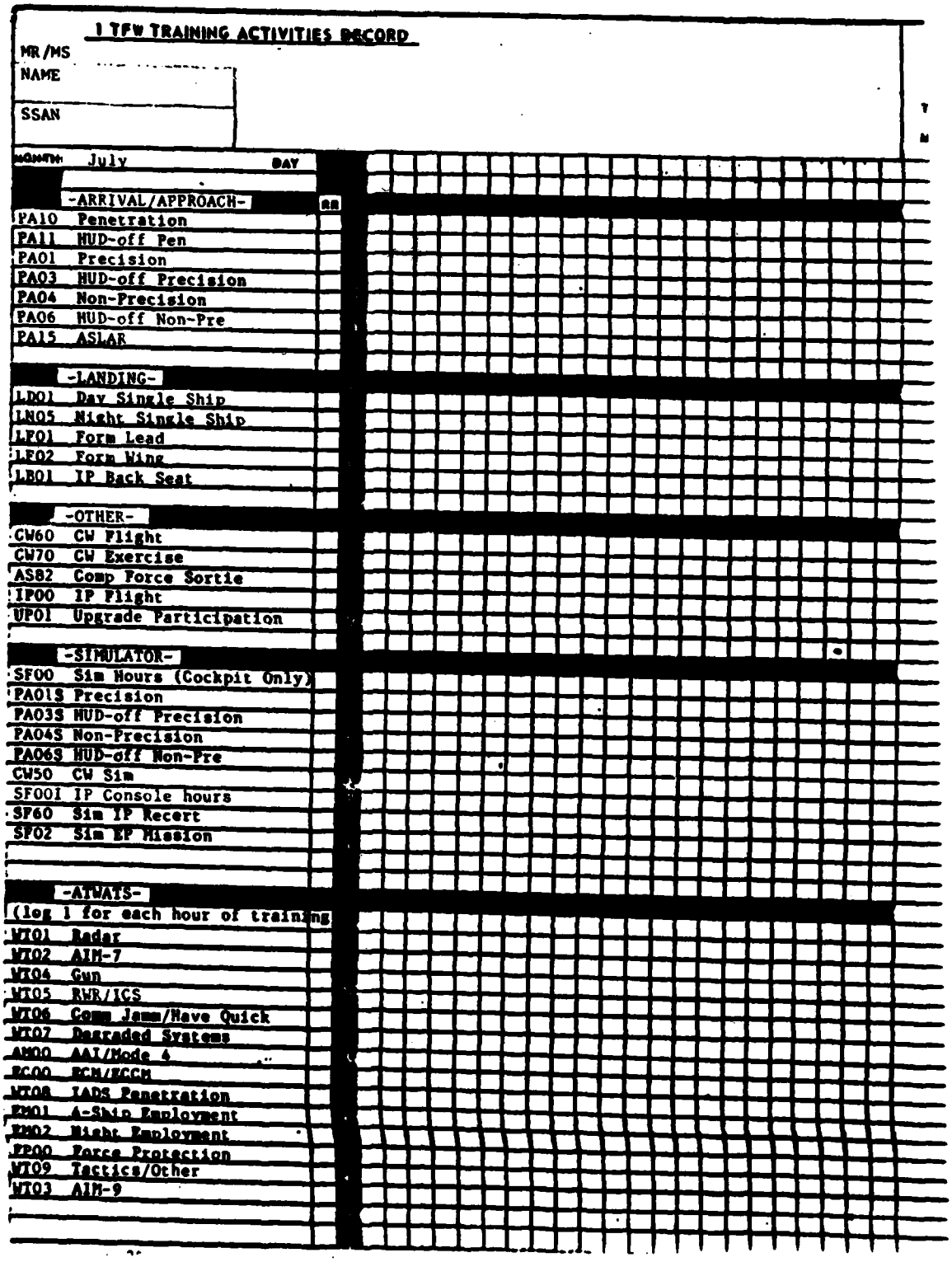
NAME

SSAN

TO & MO

MONTH: July DAY

	-Flying Time-	
	-DAY SORTIES-	RR
SG16	Intercept/Other	
SG11	AHC	
SG12	BFM	
SG13	ACM	
SG14	ACT	
SG21	DACT 2v	
SG22	DACT 4v	
SG17	DACT 2v AGG	
SG18	DACT 4v AGG	
SG27	DACT Other	
AC29	DART	
SI01	Inst Single/Chased	
SI02	Inst Dual	
SP01	Proficiency	
SC50	FCF	
SB00	IP Back Seat	
	-NIGHT SORTIES-	
SG02	A/A	
SI15	Inst Single/Chased	
SI25	Inst Dual	
SP05	Proficiency	
SB05	IP Back Seat	
	-DEPARTURE/ENROUTE-	
AC49	Alert Scramble	
TL01	Day Ld Form T/O	
TM01	Day Wg Form T/O	
TL05	Night Ld Form T/O	
TM05	Night Wg Form T/O	
AS07	Radar Trail	
AS09	Non-Radar Trail	
AR01	AAR Day	
AR05	AAR Night	
	-EVENTS/SCENARIOS-	
AC45	ACBT Events (Total #)	
WD02	Dart Hit	
AC31	LOVAT (Tgt 1000')	
AC35	Hi (Tgt 40k')	
AC05	Point Defense	
AC50	Escort	
AC51	Sweep (Other)	
AC52	Sweep (4-ship)	
AC53	CAP (Other)	
AC54	CAP (4-ship)	
	-EC EVENTS-	
EC01	Comm Jamm (5 minutes)	
EC05	ECCM Intercept	
EC07	EWR (A/A or SAM)	
EC10	EC Range Event (EWEP/LSV)	



1 TFW TRAINING ACTIVITIES RECORD

MR/MS

NAME

SSAN

MONTH: July DAY

	-ARRIVAL/APPROACH-	
PA10	Penetration	
PA11	HUD-off Pen	
PA01	Precision	
PA03	HUD-off Precision	
PA04	Non-Precision	
PA06	HUD-off Non-Pre	
PA15	ASLAR	
	-LANDING-	
LD01	Day Single Ship	
LN05	Night Single Ship	
LF01	Form Lead	
LF02	Form Wing	
LB01	IP Back Seat	
	-OTHER-	
CW60	CW Flight	
CW70	CW Exercise	
AS82	Comp Force Sortie	
IP00	IP Flight	
UP01	Upgrade Participation	
	-SIMULATOR-	
SF00	Sim Hours (Cockpit Only)	
PA01S	Precision	
PA03S	HUD-off Precision	
PA04S	Non-Precision	
PA06S	HUD-off Non-Pre	
CW50	CW Sim	
SF001	IP Console hours	
SF60	Sim IP Recert	
SF02	Sim EP Mission	
	-ATWATS-	
(log 1 for each hour of training)		
WT01	Radar	
WT02	AIM-7	
WT04	Gun	
WT05	RWR/ICS	
WT06	Comm Jamm/Have Quick	
WT07	Degraded Systems	
AM00	AAI/Mode 4	
EC00	ECM/ECCM	
WT08	IADS Penetration	
EM01	A-Ship Employment	
EM02	Night Employment	
FP00	Force Protection	
WT09	Tactics/Other	
WT03	AIM-9	

APPENDIX D

Letter of Xs

LETTER OF X's

A. REFERENCES:

1. AFM 171-190 Volumes A through J
2. AFR 55-15

B. OBJECTIVE:

Provide procedures for maintaining each squadron's letter of X's. The letter of X's will be used to document TFW pilots' qualifications as specified by the DCO to include, training status, weather category and

AS OF: 860711

TFS LETTER OF X'S

NAME FLIGHT	RNK	FI ND	E XP	W XC T	G C C L V	A A T N G	L A S D	L A U G	F L T D	F L I P	S I M P	S E F E	F C F	S O F	R O M	R O T O	AVL	REMARKS
	LTC	1	E	A	B	MR	X	X	4E					X	X	X		CC
	LTC	2	E	A	B	MR	X	X	4E					X	X	X		DO
)	MAJ	3	E	A	B	MR	X	X	4E	X*	X			X	X	X		ADO
(	MAJ	11	E	A	B	MR	X	X	4E	X*	X			X	X	X	LVAV	0907
)	CPT	12	N	B	B	MR	X								X	X		
(	CPT	13	N	D		MQ												
(	CPT	14	E	A	B	MR	X		2E		X				X	X	TDAV	1107
)	CPT	15	N	B	A	MR	T								X	X		
	1LT	18	N	C	B	MR	X								X	X		
	MAJ	21	E	A	B	MR	X	X	4E	X				X	X	X		FLCC
	CPT	23	E	B	B	MR	X		4E		X				X	X		
	CPT	24	N	C	A	MR	T								X	X		
)	CPT	25	N	D		MQ												
(	1LT	27	N	B	B	MR	X		2E		X				X	X		
	1LT	28	N	C	B	MR	X								X	X		
	CPT	29	N	A	B	MR	X								X	X		
(	MAJ	31	E	A	B	MR	X	X	4E	X	T			X	X	X		FLCC
	1LT	32	N	C	B	MR	X								X	X		
	CPT	33	E	A	B	MR	X		4		X				X	X		
	CPT	34	E	A	B	MR	X		T				T		X	X		WDOX
)	CPT	36	E	B	B	MR	X	X	4E	X	X				X	X	TDAV	1509
	CPT	37	E	B	B	MR	X		4E	T	X		X		X	X		
	1LT	38	N	C	B	MR	X								X	X	TDAV	1807
(	1LT	39	N	C	B	MR	X								X	X		
)	MAJ	41	E	A	B	MR	X		4		T			X	X	X	LVAV	0907
	1LT	42	N	C	B	MR	X								X	X		
	CPT	43	N	B	B	MR	X		T						X	X	LVAV	1407
	CPT	45	N	B	B	MR	X		4E						X	X		
(	1LT	47	E	B	B	MR	X		4E	X	X		X		X	X		
	CPT	48	N	B	B	MR	X		2E		T				X	X		
	CPT	49	E	A	B	MR	X	X	4E	X	X				X	X		
	COL	51	E	C		MS												WGCV
	MAJ	52	E	A	B	MR	X	X	4E	X	X	X		X	X	X		WDOW
)	CPT	53	E	B		MS	X	X	4E	X	X				X	X		WDOT
(	MAJ	54	E	A		MS	X		2E		X				X	X		WDOO
	CPT	55	E	A	B	MR	X	X	4E	X	X				X	X		WDOW
	CPT	56	E	A	B	MR	X	X	4E	X	X	X			X	X		WDOV
	CPT	58	E	B		MS			4									FTSG
	COL	90	E	B		MS												WGCC

E =13 FL2E 5 IP *= 2 SOF = 5
NE =16 FL4E=10 IP = 7 SEFE= 0
, LTC, USAF IQ/MQ= 2 FL/T 2 ZE = 45
IP/T= 1

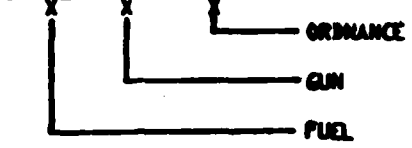
OPERATIONS OFFICER

NOTE: ABOVE FIGURES ARE FOR RPI-1 PILOTS ONLY!!!!

APPENDIX E

Scheduling Shell

F-15 CONFIGURATION CODES



FUEL CODES

A - Full Internal, No External Tanks Installed
B - Full Internal, 1 x 600 Gallon External Tank/Centerline
C - Full Internal, 2 x 600 Gallon External Tanks/Wings
D - Full Internal, 3 x 600 Gallon External Tanks/Centerline-Wings
E - 7,600 # Internal, No External Tanks Installed
F - Full Internal, Full CFT, No External Tanks
G - Full Internal, Full CFT, Centerline Tank
H - Full Internal, Full CFT, External Wing Tanks
I - Full Internal, Full CFT, Centerline/Wing Tanks
J - Full Internal, 4000 lbs CFT, No Centerline
K - Full Internal, Empty CFT, No External
L - Full Internal, Empty CFT, Full External

GUN CODES

0 - Gun/Drum Empty
1 - 500 Rounds TP/TPT
2 - 200 Rounds TP/TPT
3 - 900 Rounds TP/TPT
4 - 900 Rounds HEI

ORDNANCE

A - Pylon Station + 2 BLAU 134, Station A&B/No Ordnance
B - ACMI POD
C - AIM 9L, PTM
D - AIM 9P, PTM
E - AIM 9M, PTM
F - AN/ALQ-131 POD (C/L) Station)
G -
H -
I - Travel POD
J - NO PYLONS
K -
L -
M -
N -
O - AIM-7
P - AIM-9
Q - AIM-7/AIM-9
R - 4 x AIM 7/9 X AIM 9

NOTES

1. All AIM-9L and AIM-9P configurations (EXCEPT K and L) are captive carry missiles.

2. All AIM-9s will have repeater cables installed when loaded aircraft.

3. If only an ACMI pod is loaded (no AIM-9 PTM), a repeater cable will be installed with the pod.

4. ACMI pods should be loaded on only the outboard station 2A and 8B.

5. AIM-7 Simulator plugs will be installed to the maximum extent possible/available.

6. All aircraft will normally carry the centerline and inboard pylons on each flight except as directed.

EXTERNAL TANK INSPECTIONS:

All 600 gallon External Fuel Tanks placed in the "Tank Farm" will be inspected on a weekly basis for security.

DEFINITION OF ACRONYMS

ACTT - AIR COMBAT TACTICS TRAINING
AARD - DAY AIR-TO-AIR REFUELING
AARN - NIGHT AIR-TO-AIR REFUELING
SACM - SIMILIAR AIR COMBAT MANEUVER
DACT - DISSIMILIAR AIR COMBAT TACTICS TRAINING
DBFM - DISSIMILIAR BASIC FLIGHT MANEUVERS
SBFM - SIMILIAR BASIC FLIGHT MANEUVERS
ECCM - ELECTRONICS CONTROL-COUNTER MEASURES
DART - SELF EXPLANATORY
LCLC - LOCAL
INST - INSTRUMENT
INCP - INTERCEPT
XCRO - CROSS COUNTRY DEPART LFI REMAIN OVERNIGHT
XCOB - CROSS COUNTRY OUT AND BACK
XCRB - CROSS COUNTRY TO CROSS COUNTRY BASE
TRAN - TRANSITION
LOWL - LOW LEVEL
ADEX - AIR DEFENSE EXERCISE
DEMO - DEMONSTRATION FLIGHT
DPLY - DEPLOY TO ANOTHER LOCATION
TGTS - TARGETS
FCFA - FUNCTIONAL CHECK FLIGHT
ALRT - ALERT AIRCRAFT
PDMI - DEPOT INPUT
PDMR - DEPOT RETURN
FERR - FERRY
XCRT - CROSS COUNTRY RETURN TO LANGLEY

ADMIN - ADMINISTRATIVE FLIGHTS
LCLC - LOCAL
FCFA - FUNCTIONAL CHECK FLIGHTS
RGSP - RANGE SUPPORT
RR01 - CINCLANT
RR02 - CINCLANT
RR03 - TWA MISSION
RR04 - AIR REFUELING MISSION
RR05 - 6 ACCS TRAINING MISSION
RR06 - CROSS COUNTRY TRAINING MISSION
QBLC - QUEEN BEE LOCAL - ANDREWS AFB
HQCC - TAC COMMANDER

TABLE OF CONTENTS:

SORTIE SEQUENCE NUMBERS

101 - 110 EC-135 LCLC
111 - 119 EC-135 XOCC
201 - 299 27TH TFW
301 - 319 UH-1F LCLC
320 - 350 UH-1F XOCC
351 - 399 WILLIAM TELL
401 - 499 94TH TFS
501 - 580 27TH DEP
581 - 599 27TH XOCC
601 - 680 71ST DEP
681 - 699 71ST XOCC
701 - 799 71ST TFS
801 - 880 94TH DEP
881 - 899 94TH XOCC
901 - 999 OTHER

E-3

FM:	CM	DATE	BASE CODE	WEEKLY /DAILY AIRCRAFT FLIGHT SCHEDULE	ORGANIZATION	FLIGHT/SECTION	AIRCRAFT MDS	DAY
	A	860721	MUHJ		1 TFW	ST TFS	F-15 A/C/D	MONDAY

SORTIE SEQ NO:	AIRCRAFT ID NUMBER	SORTIE NO.	MSN	SCHEDULED TAKEOFF	SCHEDULED LANDING	SCHEDULED DURATION	CONFIG CODE	TRACK/ RANGE	ARCT/RANGE TIME	SQUADRON CALL SIGN	UX CAT	PILOT	A3034 A3037 REMARKS
701	A0059	01	AAR/ ACTT	0910	1035	01.3	BOE	AR636 386B	0925 - 0940 0940 - 1030	IRON	11		ARCT 0925-0940
702	A1053										12		
703	A3026										13		
704	A1032										14		
705	A1031	01	AAR/ DACT	0925	1040	01.3	BOEB	AR636 ACMR	0940 - 0955 1000 - 1040	IRON	31		ARCT 0940-0955
706	A1028										32		
707	A3028										33		
708	A1026										34		
709	A0058	01	AAR/ SBFM	0945	1100	01.3	BOD	AR636 386A	1000 - 1015 1015 - 1100	IRON	71		ARCT 1000-1015
710	A1029										72		
711	A0059	02	ACTT	1250	1405	01.3	BOE	386A	1300 - 1340	IRON	11		
712	A1053										12		
713	A3026										13		
714	A1032										14		
715	A1031	02	ACTT	1330	1445	01.3	BOE	386A	1340 - 1415	IRON	31		
716	A1028										32		
717	A3028										33		
718	A1026										34		

E-4

APPENDIX F

PAS Code

Notes on the Pilot Assignment (PAS) Code

The PAS code given here is for the basic daily scheduling problem. This particular version of the code is designed to be used with pruned data. In this case pruned means that only pilot qualifications for jobs which are to be assigned are stored. In addition those arcs which are infeasible due to pilot nonavailability have also been removed. This can easily be done the prior evening once the previous days flying is complete.

Input consist of the job data and the applicable (pruned) pilot qualifications. The user also specifies the print level as well as whether the job categorizing specified in the text is to be used. PAS will then return a feasible schedule if found. If not and catting was not selected, the program starts over and the user specifies catting (unless the infeasible solution presented was acceptable). Once a feasible solution is found , or an infeasible solution if catting is selected, the user is then asked whether a higher objective valued solution is desired. If so improved solutions are pursued until a single pass through all of the jobs can produce no more improvements.

The code presented here is a developmental one. One could make reductions in memory requirements as outlined in Chapter 3. Execution times can in all likelihood also be reduced significantly through some of the procedures suggested also in Chapter 3. Hopefully future research and interest by the Air Force will lead to such a code.

```
#include "timer.h"
#include "macro.h"

/*****************************************************************************

                    PILOT ASSIGNMENT ALGORITHM (PAS)

                         MARK T. MATTHEWS

                           22 JAN 1987

*****************************************************************************
*****************************************************************************/

/*************** DECLARE GLOBAL VARIABLES ************************/


/* The Node Vector */
int depth[NN]; /* depth of a node in the tree */

int pred[NN]; /* pred of a node in a tree */

int up[NN]; /* orientation of the pred arc of  a node */

int predl[NN]; /* the pred arc of a node */

int thread[NN]; /* the thread of a node */

int dual[NN]; /* dual price of a node */

int point[NN]; /* pointer to the first arc of a pilot */

int source[NN]; /* supply or demand level of a node */


/* The Arc Vector */
int cost[NA]; /* cost ( price ) of an arc */

int bnode[NA]; /* the to node of an arc */

int flow[NA]; /* flow over a arc */



/* General variables */
int n=0; /*marker for the from node */

int m=0; /*marker for the to node */

int from=0; /*marker for the from node */

int to=0; /* marker for the to node */

int jobleft=0; /* indicates whether jobs are left between transp. prob. */

int pivotcount=0; /* marks whether any pivots have occured */

int pivot=0; /* marks whether any arcs haved priced favorably */
```

```
int fmin=0; /* min flow for a ratio test */
int linkin=0; /* the entering link */
int M=10000; /* the Big M value */
int tt=0; /* counts the number of transp prob solved */
int q=0; /* The number of jobs (including sink ) */
int level=0; /* print level */
int maxl=0; /* the max number of links ( real links ) */
int N=0; /* number of pilots plus bogus */
int imp=0; /* marker indicating whether an improved solution was found */
int nogo=0; /* indicates that no feasible arcs remain for a job */
int last=0; /* general marker */
int bogus_label[NJ]; /* indicates whether that job is labeled to bogus */
int feaslinks[NJ]; /* indicates a feasible link to a given job */
int i=0; /* tracks pilot number */
int gold[NP]; /* indicates number of last job assigned */
int j_cat; /* number of job cats designated in the input file */
int cat; /* the actual cat number being priced in the algorithm */
int bflow[NA]; /* a backup tracker of arc flow */
int timeperiods=0; /* the number of timeperiods in a day */
int ttime=0; /* the number of times a transp problem has been solved */
int infeas=0; /* indicates no feasible solution found */
int price_count; /* counts the number of pricing operations */
int pivot_count; /* counts the number of pivots */
int swap_count; /* counts the number of swaps */
int imp_count; /* counts the number of improvement swaps made */


/* double variables */
double endjob=0.0; /* indicates when the the last job ends */
double jobstart=10000.0; /* indicates when the first job starts */
double fillindex[NJ]; /* the fillindex of a given job */
double crewrest=0.0; /* the length of the crewrest period */
```

```
/* structs  ( see "macro.h" ) */
JV job[NJ]; /* struct for each job */

JV *job_pointer[NP]; /* pointer to each pilot assigned job struct */

JV pilot[NP]; /* struct for each pilot */

HP heap[NJ];  /* struct for each job in the heap */

HP *hp; /* pointer to each job heap struct */



/***************** BODY OF PROGRAM " MAIN " ************************/



main ()

 {

/******************* DECLARE LOCAL MAIN VARIABLES ****************/


/* general utility variables */
  int na,nn,oldc,work,bprice,old,aa,bb,cc,k,totcost=0,can_do;

  int cont,cat_start=0,cs,trip=0,uj,date[4];


  int found_label; /* indicates a job is labeled to "bogus"   */


/* structs */
  JV n_in;


/* time tracker variables */
  double data_in_time, source_time, init_time, price_time, pre_swap_time;

  double pivot_time, assign_time, change_time, swap_time, infeas_time;

  double heap_time, impr_time, tot_time, stop;



/******************** Initializing ******************************/

/*  Set the number of timeperiods in a day */
  start_over: timeperiods = 100;

  pre_swap_time = 0.0;


/* Set the length of the crewrest periods */
  crewrest = .0050;


/* Input the data */
```

```
  get_time(&rstart);

  data_in ();

  get_time(&rstop);
  cpu_time = show_time(&rstart,&rstop);
  data_in_time += cpu_time.usr;


/* Initialize the total cost    */
  totcost = (q*M) + 1;


/* If "catting" not selected look at all jobs else look at jobs by cat */
if ( infeas == 0 )
  cat_start = j_cat;
else
  cat_start = 1;

for ( cat=cat_start; cat <= j_cat; ++ cat )
  {


   i = 0;


/* update arc prices based on previous job assignments */
   if ( cat != 1 )
    change_cost ();


/* update job deamands based on previous job asignments */
  repeat: get_time(&rstart);
  cpu_time_start = show_time(&rstart,&rstop);

  jobsource ();

  get_time(&rstop);
  cpu_time_stop = show_time(&rstart,&rstop);
  source_time += (cpu_time_stop.usr - cpu_time_start.usr);
  tot_time += cpu_time_stop.usr - cpu_time_start.usr;


/* initialize using the Big M method */
  get_time(&rstart);
  cpu_time_start = show_time(&rstart,&rstop);

  initialize ();

  get_time(&rstop);
  cpu_time_stop = show_time(&rstart,&rstop);
  init_time += (cpu_time_stop.usr - cpu_time_start.usr);
  tot_time +=  cpu_time_stop.usr - cpu_time_start.usr;



/********************* Pricing and Pivoting ***************************/


   i = 0;

  while ( i<= N )

   {
```

```
/* Price each eligible arc */
  get_time(&rstart);
  cpu_time_start = show_time(&rstart,&rstop);

  price ();

  get_time(&rstop);
  cpu_time_stop = show_time(&rstart,&rstop);
  price_time += (cpu_time_stop.usr - cpu_time_start.usr);
  tot_time +=  cpu_time_stop.usr - cpu_time_start.usr;


/* If an arc prices favorably pivot it in */
  if ( pivot < 0 )
   {

  get_time(&rstart);
  cpu_time_start = show_time(&rstart,&rstop);

  pivot_out ();

  get_time(&rstop);
  cpu_time_stop = show_time(&rstart,&rstop);
  pivot_time += (cpu_time_stop.usr - cpu_time_start.usr);
  tot_time +=  cpu_time_stop.usr - cpu_time_start.usr;

    }
 /* if an arc priced favorably go through the arcs again */
  if ( i==N && pivotcount ==1 )

     {

      i = 0;
      pivotcount = 0;

     }

   }


/* assign pilots jobs based on the results of the simplex algorithm */
  get_time(&rstart);
  cpu_time_start = show_time(&rstart,&rstop);

  assign ();

  get_time(&rstop);
  cpu_time_stop = show_time(&rstart,&rstop);
  assign_time += (cpu_time_stop.usr - cpu_time_start.usr);
  tot_time +=  cpu_time_stop.usr - cpu_time_start.usr;


/* if there are any jobs left do the following */
  if ( jobleft == 1 )

    {

     can_do = 0;


/* update arc prices based on job assignments */
  repeat2: get_time(&rstart);
  cpu_time_start = show_time(&rstart,&rstop);

  change_cost ();
```

```
  get_time(&rstop);
  cpu_time_stop = show_time(&rstart,&rstop);
  change_time += (cpu_time_stop.usr - cpu_time_start.usr);
  tot_time +=  cpu_time_stop.usr - cpu_time_start.usr;


/* See if any feasible links exist to unassigned jobs */
    while ( job_pointer[N]->number != 0)
     {

       if ( feaslinks[job_pointer[N]->number] == 1 )
        can_do = 1;

       job_pointer[N] = job_pointer[N]->next;

     }

    job_pointer[N] = &pilot[N].assigned[1];


/*****************.******* Swapping ****************************************/

/* If no feasible links exist try to swap out the job */
    if ( can_do == 0 )
     {

get_time(&rstart);
cpu_time_start = show_time(&rstart,&rstop);

if (swap_count == 0 )
 pre_swap_time = price_time;

      swap (N);

get_time(&rstop);
cpu_time_stop = show_time(&rstart,&rstop);
swap_time += cpu_time_stop.usr - cpu_time_start.usr;
tot_time += cpu_time_stop.usr - cpu_time_start.usr;
    if ( nogo != 1 )
      goto repeat2;

     }

/* otherwise start over with the reduced transportation problem */
    else if ( can_do == 1 || ( nogo != 1 && can_do == 0 ) )
     {

      zero_pilotvector(N);
      goto repeat;

     }

   }



}
```

```
printf ("Initial soln usr time is %6.2f \n", tot_time);

/* make the final initial solution assignments */
get_time(&rstart);
cpu_time_start = show_time(&rstart,&rstop);

 assign (N);

get_time(&rstop);
cpu_time_stop = show_time(&rstart,&rstop);
assign_time += cpu_time_stop.usr - cpu_time_start.usr;
tot_time += cpu_time_stop.usr - cpu_time_start.usr;



/* find the jobs labeled to bogus */
get_time(&rstart);
cpu_time_start = show_time(&rstart,&rstop);

job_pointer[N] = &pilot[N].assigned[1];

for (i=1; i<=q; ++i)
   if (bogus_label[i] == 1)
     {

     found_label = 0;
     while (job_pointer[N]->number != 0)
      {

       if (job_pointer[N]->number == i)
         {

           job_pointer[N] = pilot[N].assigned[gold[N]].next;
           found_label = 1;

         }
        else if (job_pointer[N]->next->number == 0 && found_label == 0)
         {

           n_in = job[i];
           n_in.number = i;
           add(N,n_in);
           job_pointer[N] = pilot[N].assigned[gold[N]].next;

         }
       else
         job_pointer[N] = job_pointer[N]->next;

      }

     job_pointer[N] = &pilot[N].assigned[1];

     }

job_pointer[N] = &pilot[N].assigned[1];



/* If any jobs are labeled to bogus print who is qualified to do it */
while (job_pointer[N]->number != 0)
 {
   uj = job_pointer[N]->number;
```

```
  printf("\n");
  printf("Infeasible soln found");
  printf("\n");

if ( level == 10 || level == 7 )
 {
  printf ("Unable to assign job %d\n", job_pointer[N]->number);
  printf ("The following pilots can perform this job\n");
  printf ("Pilot#    Job#    Start      Stop\n\n");
  for (i=1; i<N; ++i)
  {
   for (aa=1; aa<=pilot[i].next; ++aa)
     if (pilot[i].type[aa] == job[uj].type)
       while (job_pointer[i]->number != 0)
         {

          printf ("%3d%10d",i,job_pointer[i]->number);
          for (k=1; k<=3; ++k)
            date[k]=convert(job_pointer[i]->start,k);
          flug: if(date[3] <100)
            printf ("      00%2d",date[3]);
          else if ( date[3] < 1000 && date[3]>= 100 )
            printf ("      0%3d",date[3]);
          else
            printf ("      %4d",date[3]);

          if (trip != 1)
            {

             stop = job_pointer[i]->length + job_pointer[i]->start;
             for (k=1; k<=3; ++k)
               date[k]=convert(stop,k);
             trip = 1;
             goto flug;

            }
          else
            {

             trip = 0;
             printf("\n");

            }

          job_pointer[i] = job_pointer[i]->next;

         }

        job_pointer[i] = &pilot[i].assigned[1];

       }

     }

     job_pointer[N] = job_pointer[N]->next;

    }

  job_pointer[N] = &pilot[N].assigned[1];



/* If we're infeasible and "catting" not selected, exit so user can start over
   with  "catting "                                                          */
```

```
if ( job_pointer[N]->number != 0 && infeas == 0)
  {

   printf("Start over\n\n");
   exit(9);

  }


get_time(&rstop);
cpu_time_stop = show_time(&rstart,&rstop);
infeas_time += cpu_time_stop.usr - cpu_time_start.usr;
tot_time +=  cpu_time_stop.usr - cpu_time_start.usr;


/************************** Improving ***********************************/

doit: get_time(&rstart);
cpu_time_start = show_time(&rstart,&rstop);



/* order jobs in ascending order based on arc prices */
heap[0].arc = 0;
bnode[0] = N;
heap[0].cost = M+1;
heap[0].next = &heap[NJ-1];
heap[NJ-1].arc = NA-1;
bnode[NA-1] = 0;
heap[NJ-1].cost = -M-1;
heap[NJ-1].next = &heap[0];

for (aa=1; aa<=N; ++aa)
for (k=point[aa]; k<=point[aa+1]-1; ++k)
  {

    old = NJ-1;
    hp = &heap[0];
    if ( bflow[k] > 0 && bnode[k]-N != q)
      {

        heap[bnode[k]-N].arc = k;
        for (cc=1; cc<=pilot[aa].next; ++cc)
        if(pilot[aa].type[cc]==job[bnode[k]-N].type)
          {

            heap[bnode[k]-N].cost = pilot[aa].cost[cc];
            heap[bnode[k]-N].pilot = aa;
            while (heap[bnode[k]-N].cost<=hp->cost )
              {

                heap[bnode[k]-N].next = hp->next;
                heap[bnode[hp->arc]-N].next = &heap[bnode[k]-N];
                heap[old].next = &heap[bnode[hp->arc]-N];
                old = bnode[hp->arc]-N;
                hp = heap[bnode[k]-N].next;

              }

          }
      }
}
```

```
/* compute total present cost */
hp = heap[0].next;
oldc = totcost;
totcost = 0;
while ( hp->arc != NA -1)

  {

    bprice = hp->cost;
    work = bnode[hp->arc]-N;
    if ( level == 10 )
      printf("%5d  %5d\n",work,bprice);
    totcost += bprice;
    hp = hp->next;

  }

get_time(&rstop);
cpu_time_stop = show_time(&rstart,&rstop);
heap_time += cpu_time_stop.usr - cpu_time_start.usr;
tot_time +=  cpu_time_stop.usr - cpu_time_start.usr;


printf ("Total cost is %d\n",totcost);


printf("Do you want to get an improved solution?(yes = 1)\n\n");
scanf("%d", &cont);

if ( cont == 0 )
   exit(8);

/* if totalcost has improved try to find another improvement */
if (totcost < oldc )
{

hp = heap[0].next;
while (hp->arc != NA-1)
   {

     aa = bnode[hp->arc]-N;
     bb = hp->pilot;
     cs = hp->cost;

get_time(&rstart);
cpu_time_start = show_time(&rstart,&rstop);

      impr(aa,bb,cs);

get_time(&rstop);
cpu_time_stop = show_time(&rstart,&rstop);
impr_time += cpu_time_stop.usr - cpu_time_start.usr;
tot_time +=  cpu_time_stop.usr - cpu_time_start.usr;

      hp = hp->next;

   }

     if (imp ==1)
      {

get_time(&rstart);
cpu_time_start = show_time(&rstart,&rstop);
```

```
      assign ();

get_time(&rstop);
cpu_time_stop = show_time(&rstart,&rstop);
impr_time += cpu_time_stop.usr - cpu_time_start.usr;
tot_time +=  cpu_time_stop.usr - cpu_time_start.usr;

         imp = 0;

printf ("Tot time to this impr is %6.2f \n", tot_time);
goto doit;



       }

}



/********************** Data printout *********************************/

/* compute and print the problem stats */
printf("\n\n");
printf ("Total usr time is %6.2f \n",tot_time);

printf("\n");
nn = N +q +1;
na = point[N+1]-1;
printf("%d Nodes  %d Arcs\n",      nn, na);
printf("Price_count is %d\n", price_count);
printf("Pivot_count is %d\n", pivot_count);
printf("Swap_count  is %d\n",  swap_count);
printf("Imp_count   is %d\n\n\n",   imp_count);

printf("Input time is %6.2f\n",data_in_time);
printf("Source time is %6.2f\n",source_time);
printf("Init time is %6.2f\n",init_time);

if (swap_count != 0 )
  price_time -= pre_swap_time;
else
  {
   pre_swap_time = price_time;
   price_time = 0.0;
  }
printf("Price time is %6.2f (preswap) and %6.2f (postswap)\n",pre_swap_time,
       price_time);

printf("Pivot time is %6.2f\n",pivot_time);
printf("Assign time is %6.2f\n",assign_time);
printf("Change time is %6.2f\n",change_time);
printf("Swap time is %6.2f\n",swap_time);
printf("Infeas time is %6.2f\n",infeas_time);
printf("Heap time is %6.2f\n",heap_time);
printf("Improve time is %6.2f\n",impr_time);

  }



/******************* Last Card of the Main program ************************/
```

```
#define NP 51  /* number of pilots */
#define NJ 51  /* number of jobs   */
#define NA 1001 /* number of arcs */
#define NQ 21  /* numer of quals per pilot */
#define ND 11 /* max number of duties assigned a pilot */
#define NN 101 /* total number of nodes */
#define MJT 1001 /* the highest number used for a given job type */


/* define the job struct  */
typedef  struct  jobvector {

      int       cat; /* the job category */
      int       type; /* the job type */
      int       number; /* the number (demand) of a job */
      int       label;  /* marker indicating whether the job has been labeled */
      int       con[NJ]; /* the conflicting jobs */
      double    start; /* job start time */
      double    length; /* the length of a job */
      struct jobvector *next; /* pointer to next job */

     }JV;


/* define the pilot struct */
typedef struct  {

      int type[NQ]; /* the job types a pilot is qualified to perform */
      int cost[NQ]; /* a pilot's price to do a job */
      int swapout[ND]; /* tracks jobs a pilot must swap to take another job */
      JV assigned[ND]; /* the jobs assigned to a pilot */

     }PV;


/* define the struct for the heap */
typedef struct heaper {

    int arc; /* the arc of an assigned job */
    int cost; /* the above assigned arc cost */
    int pilot; /* the pilot a job is assigned to */
    struct heaper *next; /* pointer to the next job in the heap */

   }HP;
```

```
#include <sys/time.h>
#include <sys/resource.h>

typedef struct {  /* for accumulation user and system time */
  double     usr, sys;
} CPU_TIME;

get_time( rval )
  struct rusage    *rval;
{
  getrusage( RUSAGE_SELF, rval );
}

CPU_TIME  show_time( t0, t1 )
  struct rusage    *t1, *t0;
{
  CPU_TIME    t;

  t.usr = (double)(t1->ru_utime.tv_usec - t0->ru_utime.tv_usec);
  t.usr /= 1000000;
  t.usr += (double)(t1->ru_utime.tv_sec - t0->ru_utime.tv_sec);

  t.sys = (double)(t1->ru_stime.tv_usec - t0->ru_stime.tv_usec);
  t.sys /= 1000000;
  t.sys += (double)(t1->ru_stime.tv_sec - t0->ru_stime.tv_sec);
  return( t );
}
```

```
#include <sys/time.h>
#include <sys/resource.h>

typedef struct {  /* for accumulation user and system time */
  double     usr, sys;
} CPU_TIME;

int         get_time();     /* has one argument of type struct rusage */
CPU_TIME    show_time();    /* two arguments, returns cpu_time */

static struct rusage        rstop, rstart;
static CPU_TIME             cpu_time,cpu_time_start,cpu_time_stop;
```

```
#include <stdio.h>
#include "macro.h"

/*************** BODY OF DATA_IN PROGRAM ******************************/

/*This routine takes two input files. The first file is the job file such as
the following example:

    1  110   1.126292e+01   2.083333e-03    1
    1  120   1.126292e+01   2.083333e-03    1
    1  130   1.126292e+01   2.083333e-03    1
    1  140   1.126417e+01   4.166667e-04    1
    2  210   1.126250e+01   2.500000e-03    1
    3  310   1.126250e+01   2.500000e-03    1
    4  410   1.126250e+01   2.500000e-03    1
    4  410   1.126333e+01   2.500000e-03    1
    4  420   1.126333e+01   2.500000e-03    1
    5  510   1.126250e+01   2.500000e-03    1
    5  510   1.126333e+01   2.500000e-03    1

The first column is the category number, the second the job type, the third
is job start time, the fourth job length, and the fifth is the number of jobs
of this type occuring at this time. The first two digits of job start time
represent the month, the second two the day, the rest of the digits are the
time of day where 1.000000e-02 is 24 hours.

The second file is the pilot qualification file such as the following:

    1  110  -27
    1  120  -56
    1  130  -57
    1  140  -83
    1  410   -4
    1  420  -57
    1  510  -36
    1  610  -99
    2  110  -62
    2  120  -42
    2  130  -22
    2  140  -96
    2  310  -63
    2  410  -14
    2  420  -54
    2  510  -11
    2  610  -20

The first column is the pilot number, the second is the job type and the third
is the benefit of that pilot performing that job type. This file has already in
fact been synthesized to account for pilot nonavailability and to sequentially
number the pilots for input (in a separate routine not shown here ).

These files combine to form a third data listing which contains the anode,bnode,
and cost (price) for each arc.                                              */


data_in ()

{

/****************** Declare Global Variables ******************************/

    extern int j_cat,feaslinks[],M,level,q,maxl;

    extern int infeas,N,point[],source[],cost[],bnode[];
```

```
    extern double crewrest,jobstart,endjob,fillindex[];
    extern JV job[],*job_pointer[];
    extern PV pilot[];

/****************** Declare Local Data_in Variables ************************/
int totjob=0; /* the total number of jobs */
int anode[3]; /* the from node */
double sumjob[MJT]; /* the total number of a particular jobtype */
double pqnumber[MJT]; /* the number of pilots qualified to perform a jobtype */
int aa,bb,i,j,k,x,eof_flag=5; /* utility variables */


/*Char variables use to read in file names*/
    char in_name1[25],in_name2[25];
    FILE *in_file1, *in_file2, *fopen();



/* initialize variables */
    anode[1]=0; anode[2]=0; anode[3]=0;
    i = 0;

/* initialize job_pointers */
    for ( aa=0; aa<=NP-1; ++aa )
     {
      job_pointer[aa] = &pilot[aa].assigned[1];
      for ( bb=0; bb<ND-1; ++bb )
        pilot[aa].assigned[bb].next = &pilot[aa].assigned[bb+1];
     }


/*********************** Opening the Files ***********************************/

    printf ("Enter the name of the job file:\n\n");
    scanf ("%24s", in_name1);
    printf("\n");

    printf ("Enter the name of the pilot file:\n\n");
    scanf ("%24s", in_name2);
    printf ("\n\n");

    printf ("Do you want to cat infeas? (0 no,  1 yes) :\n\n");
    scanf ("%d", &infeas);
    printf ("n\n");
```

```
    printf ("The following print levels are available:\n");
    printf ("    2. Data read in files\n\n");
    printf ("    4. Initialization\n\n");
    printf ("    5. Pricing\n\n");
    printf ("    6. Pivot\n\n");
    printf ("    7. Assign\n\n");
    printf ("Enter the print level\n\n");
    scanf ("%d", &level);
    printf("\n\n");

    in_file1 = fopen (in_name1, "r" );
    in_file2 = fopen (in_name2, "r" );

    if ( in_file1 == NULL )
      {

       printf ( "couldn't open %s for reading.\n", in_name1);
       exit (2);

      }

    if ( in_file2 == NULL )
      {

       printf ( "couldn't open %s for reading.\n", in_name2);
       exit (2);

      }

    jobstart = M;

    endjob = 0;

    k = 1;

    j_cat = 0;

    if (level== 2 )
     printf ("The job file is:\n\n");


/* Read in the job file data */
    while ( eof_flag != EOF )
     {

      eof_flag = fscanf( in_file1,"%d", &job[k].cat);

      if ( j_cat < job[k].cat )
        j_cat = job[k].cat;


      fscanf ( in_file1,"%5d%15e%15e%5d", &job[k].type, &job[k].start,
               &job[k].length, &job[k].number);

      if (job[k].start < jobstart && job[k].start != 0)
       jobstart = job[k].start;

      if (job[k].start + job[k].length > endjob )
       endjob = job[k].start + job[k].length;

      if ( eof_flag != EOF )
       {

        if ( level == 2 )
         printf("%5d%5d%15e%15e%5d\n",job[k].cat,job[k].type,job[k].start,
                job[k].length,job[k].number);
```

```
        sumjob[job[k].type] += job[k].number;
        totjob += job[k].number;

       }

      k += 1;

     }


/* Set the sink job type */
    job[k-1].type = 999;

    q = k-1;

  if ( level == 2)
    {

     printf("%5d%5d%15e%15e%5d\n\n",job[q].cat,job[q].type,job[q].start,
           job[q].length,job[q].number);

     printf ("The first job starts at %e\n\n",jobstart);
     printf ("The last job ends at %e\n\n",endjob);

    }

    k=1; j=1; i=1;


/* establish the job interference sets */
    while ( job[j].cat != 0 )
     {

      if ( level==2)
       {

        printf ("\n\n");
        printf ("For job %d interference is:\n",j);

       }

      while (job[k].cat != 0 )
       {

        if ((job[j].start + job[j].length > job[k].start  &&
            job[j].start < job[k].start + job[k].length )

                          ||

           (job[j].start + crewrest < job[k].start + job[k].length)

                          ||

           (job[j].start + job[j].length > job[k].start + crewrest )  )

          {

           job[j].con[i] = k;
           if (level==2)
            printf ("%3d%3d%3d\n",j,i,job[j].con[i]);
           i += 1;

          }

         k += 1;
```

```
        }

       j += 1;

       k=1; i=1;

      }

   k = 1; i=1; j=1;

   eof_flag = 3;




/* Read in the pilot qualification data */
   while ( eof_flag != EOF )
    {

     eof_flag = fscanf (in_file2,"%5d",&anode[2]);
     fscanf ( in_file2,"%5d%5d", &pilot[anode[2]].type[k],
              &pilot[anode[2]].cost[k]);

     if ( eof_flag != EOF )
      pqualified[pilot[anode[2]].type[k]] += 1;

     if ( anode[2] > anode[1] )
       {

        pilot[anode[2]].type[1] = pilot[anode[2]].type[k];
        pilot[anode[2]].cost[1] = pilot[anode[2]].cost[k];
        pilot[anode[1]].next = k;
        pilot[anode[1]].type[k] = job[q].type;
        pilot[anode[1]].cost[k] = 0;
        source[anode[2]] = 1;
        anode[1] = anode[2];
        N += 1;
        k = 1;

       }

      k += 1;

     }

   pilot[anode[2]].type[k-1] = job[q].type;
   pilot[anode[2]].cost[k-1] = 0;
   pilot[anode[2]].next = k-1;
   if ( level ==2 )
    printf ("The fillindexes are:\n\n");
   x = 1;



/* Compute the fillindexes for each job */
   for ( k=1; k<=q; ++k )
    {

     if ( job[k].type != job[k-1].type )
      {

       pilot[anode[2]+1].type[x] = job[k].type;
       pilot[anode[2]+1].cost[x] = M;
```

```
        if ( k==q )
          pilot[anode[2]+1].cost[x] = 0;
        x += 1;

       }

      if (sumjob[job[k].type] != 0 )

       fillindex[k] = pqnumber[job[k].type]/sumjob[job[k].type];

      if (level == 2 )
       printf("job %d...  %e\n",k,fillindex[k]);

     }

    pilot[anode[2]+1].next = x-1;
    N += 1;



/* Set the supply level for bogus */
    source[N+q] = -N+1;
    if ( level == 2 )
     {

      printf ( "\n\n" );
      printf ( "The pilot file is:\n\n");
      for ( i=1; i<=N; ++i )
       for (k = 1; k <= pilot[i].next; ++k )
        printf ("%5d%5d%5d\n",i,pilot[i].type[k],pilot[i].cost[k] );

      }


    printf("\n\n");
    eof_flag = 3; anode[1] = 0; anode[2] = 0; i = 0;

    x=0;



/* Determine the anode, bnode, and arc price data */
    for ( i=1; i<=N; ++i )
     {

      j=0;
      point[i] = x+1;
      for ( k=1; k<=pilot[i].next; ++k )
       while ( job[j].type <= pilot[i].type[k] && j <= q)
         {

          if ( job[j].type == pilot[i].type[k] )
           {

            x += 1;
            cost[x] = pilot[i].cost[k];
            bnode[x] = N+j;
            feaslinks[bnode[x]-N] = 1;

           }

          j += 1;
```

```
        }

      }

   maxl = x; point[N+1] = maxl + 1;

  if (level==2)

    {

     printf ("\n\n");
     for (i=1; i<=N; ++i)

      {

       printf ("Pilot's %d arcs are:\n\n",i);
       for (k=point[i]; k<=point[i+1]-1; ++k)
        printf ("%5d%5d%10d\n",k,bnode[k],cost[k]);

      }

     printf ("\n\n");
     printf (" There are %d job-times (including nothing job)\n",q);
     printf (" There are %d total real jobs\n",totjob);
     printf (" There are %d total pilots (including bogus)\n\n",N );

    }


fclose (in_file1);
fclose (in_file2);


/****************** Last card of Data_in *********************************/

}
```

```
#include "macro.h"


/******************** Body of Program Jobsource **********************/

/*************************************************************************

Jobsource updates the demand level of jobs based on current job assignments

*************************************************************************/



jobsource ()

{


/*************** Declare global variables *************************/


extern int infeas,cat,point[],bnode[],flow[],source[],jobleft,q,N;

extern JV job[];

/*************** Declare local variables ************************/

int i;

/* Set the source values for each job */
if (jobleft == 0)
 for ( i=1; i<q; ++i )
   if (job[i].cat == cat || cat == 0 || infeas == 0)
  {

   source[N+i] = -job[i].number;
   source[N] += job[i].number;

  }
else

   source[N+i] = 0;

/* Between transportation problems update job demand levels based on pre
vious job assignments                                                   */
if (jobleft ==1 )
 {

  jobleft = 0;
  source[N] = 0;
  i = point[N];

  while ( bnode[i] < N+q  )
    {

     if ( job[bnode[i]-N].cat <= cat || cat == 0 || infeas == 0)
       {

        source[bnode[i]] = -flow[i];
        source[N] += flow[i];
```

```
            flow[i] = 0;
          }
        i += 1;
      }
    }
/********************* Last card of jobsource ************************/
}
```

```
#include "macro.h"


/****************** Body of the Change_cost Program ********************/

/***********************************************************************
Change_cost updates the arc prices based on current job assignments
***********************************************************************/


change_cost ()

{

/***************** Declare Global Variables ******************************/

   extern int ttime,q,feaslinks[],level,N,M,point[],bnode[],cost[];

   extern JV job[],*job_pointer[];

   extern PV pilot[];


/*************** Declare Local Change_cost Variables ********************/

   int i,j=0,x,k,z;


/* Count the number of the transporattion problem being solved */
 ttime +=1;


 for (i=1; i<N; ++i)
  {

   if (level==8)
    printf("For pilot %d\n",i);


/* Look at each job assigned to a pilot */
   while ( job_pointer[i]->number != 0 )
     {

      if ( level==8 )
       printf ("Job %d\n",job_pointer[i]->number);

      x = point[i];
      k=1;


/* Look at each job that an assigned job intereferes with and change the arc.
   Change the arc price on interefering ( infeasible ) arcs.                */
      while ( job[job_pointer[i]->number].con[k] != 0 )
        {
```

```
            z= job[job_pointer[i]->number].con[k] + N;
            if ( level==8)
             printf("this job int. with node%d\n\n",z);

            while ( x<=point[i+1]-1 && bnode[x] <= z )
              {

               if ( bnode[x] ==  z )
                cost[x] = ttime+M;
               else if (cost[x] != ttime+M)
                {

                for(j=1; j<=pilot[i].next; ++j)
                 if(pilot[i].type[j] == job[z-N].type)
                   {

                    cost[x] = pilot[i].cost[j];
                    break;

                   }
                }

               if  (level==8)
                 printf ("bnode[x] %d, z is %d, and cost[x] is %d\n\n",
                      bnode[x],z,cost[x]);
               x += 1;

              }

           k += 1;

        }

      job_pointer[i] = job_pointer[i]->next;

      }

    job_pointer[i] = &pilot[i].assigned[1];

}

for ( i=1; i<=q; ++i )
 feaslinks[i] = 0;
/* Determine if any feasible links exist to a particular job */
for (i=1; i<point[N]; ++i)
  {

   if (cost[i] < M && bnode[i] != N+q)
    {

     feaslinks[bnode[i]-N] = 1;
     if (level==8)
      {

       x = bnode[i] -N;
       printf ("Feaslink for %d exists with link %d\n\n",x,i);

      }

     }

    }


 if ( level == 8 )
   {
```

```
    for ( i=1; i<=N; ++i)
      for ( x=point[i]; x<=point[i+1] -1; ++x )
        printf ("%3d%4d%7d\n",i,bnode[x],cost[x] );

    }


/**************** Last card of Change_cost ****************************/

}
```

```
/************** Body of the Initialize Program ************************/

/*********************************************************************

This routine creates an initial solution using the Big M method

*********************************************************************/


initialize ()

{

/********************* Declare Global Variables ***********************/

    extern int level,q,maxl,fmin,tt,M,N;

    extern int source[],point[],dual[],pred[],predl[];

    extern int up[],thread[],depth[],cost[],flow[];

/******************* Declare Local Variables ************************/

    int k,i;

    printf ("Initializing\n");

    tt = N + q + 1;
    point[tt] =  maxl + 1;
    point[tt+1] =  maxl + N + q + 1;

    /* This is part of the cat mod */
    for(k=1; k<=point[tt+1]; ++k)
      flow[k] = 0;

/*****************************************************************************
*                                                                            *
*                        ASSIGNING INITIAL VALUES                            *
*                                                                            *
*****************************************************************************/


    /* Assign the initial flows (find an initial basis ) using the big M

       method. Construct the initial tree with this basis.              */


    fmin = M;
    depth[tt] = 0; pred[tt] = 0; predl[tt] = 0; dual[tt] = 0; thread[tt] = 1;

    for (i=1; i<=N+q; ++i )
     {

      thread[i] = i+1;
      pred[i] = tt;
      predl[i] = maxl + i;
      depth[i] = 1;
      if ( source[i] > 0 )
       {
```

```
        up[i] = 1;
        dual[i] = -M-1;
        flow[maxl+i] = source[i];

       }
      else
       {

        dual[i] = M+1;
        flow[maxl+i] = -source[i];

       }

      cost[maxl+i] = M+1;

     }


     if  ( level==4)
      {

       k =0;
       printf ("Pilot %d arcs are:\n\n",tt);
       for (i=point[tt]; i<=point[tt+1] -1; ++i)
        {

         k +=1;
         printf ("%5d%5d%5d%10d\n\n",i,k,flow[i],cost[i] );

        }


     printf("Node Source Depth Thread Pred Predl Up     Dual\n\n");
     for (i=1; i<=tt; ++i )

      {

       printf ("%3d%6d%7d%7d%6d%5d%5d%8d\n",
               i,source[i],depth[i],thread[i],pred[i],predl[i],up[i],dual[i]);

       }

     }


/******************** Last Card of Initialize  ********************************/
  }
```

```
/********************** Body of Price Program ******************************/
/*****************************************************************************
Price looks for entering arcs using the row most negative rule
*****************************************************************************/



price ()
{
/******************* Declare Global Variables ******************************/

 extern int i, point[],dual[],cost[],bnode[],linkin,from,to;
 extern int N,M,price_count,pivot,pivotcount,level;

/******************* Declare Local Variables *******************************/
 int cbar; /* the reduced cost of an arc */
 int l; /* the actual arc number */


/*****************************************************************************
*                                                                            *
*                                  PRICING                                   *
*                                                                            *
*****************************************************************************/


       i += 1;

       pivot = 0;

/* Look for the most negative reduced cost on a link out of node i.  This
   will be the link that we will pivot in. ( Row most negative rule ).
   If we have no negative reduced cost we are optimal.                  */


    for ( l = point[i]; l<= point[i+1] -1; ++l )
      if ( cost[l] < M || i == N)
        {

         price_count += 1;

         cbar = dual[i] + cost[l] - dual[bnode[l]];
```

```
        if ( cbar <  pivot )
         {
          pivotcount = 1;
          pivot = cbar;
          linkin = 1;
          from = i;
          to = bnode[1];
         }
       }
   if ( level == 5 )
    printf("Cbar is %d and linkin is %d\n", cbar, linkin);


/***************** Last Card of Price ******************************/
}
```

```
/***************** Body of Pivot_out Program ******************************/

/*************************************************************************

Pivot_out finds the leaving arc using a standard ratio test and then updates
the basis by rehanging the tree

*************************************************************************/



pivot_out ()

{

/********************** Declare Global Variables ***************************/

 extern int tt,n,m,from,to,source[],depth[],pred[],up[],flow[],predl[];

 extern int pivot_count,N,level,fmin,linkin,thread[],dual[],cost[],bnode[],M;


/********************  Declare Local Variables ****************************/

 int A,B,C,D,E,F,last,l1,l2,z; /* utility variables */

 int nmin; /* the node whose predecessor arc is the arc to leave the basis */

 int cut; /* indicates whether cut is on the up or down side */

 int jnode; /* the joining node of a cycle formed by the entering arc */

 int linkout; /* the leaving arc */

 int oldepth,chdepth,chdual; /* respectively the old depth of a node, the change


 /*************************************************************************
 *                                                                       *
 *                              PIVOTING                                 *
 *                                                                       *
 *************************************************************************/

      /* Find the link to leave the basis. The link that will leave will be
         either an "up" link on the down side (side of the from node of the
         entering link) of the tree or a "down" link on the upside (side of
         the to node of the entering link)  of the tree. The particular choice
         of which link to leave is decided by the link which is the closest
         to its lower bound (0). In other words we are doing a ratio test.
         The entering link must enter with flow > or = to 0 thus we must
         maintain flow conservation on the links in the cycle formed in
```

```
   the tree by this entering link. Since we can't have negative flows
   the link that goes to zero first leaves. More than one link could
   have its flow go to zero. To resovle ties and avoid the possibility
   of cycling an implicit perturbation method is used such that ,in
   case of  tie, the highest link on the downside or the lowest link
   on the upside leaves. This is done by saying flow[predl[m]] must
   be < fmin while flow[predl[n]] must be <= fmin.                */

pivot_count += 1;

n = from;
m = to;

/* First find the deepest node of the from and to node of the enter-
   ing link and then check its predlinks for the minimum flow in the
   cycle until you get to the same depth as the higher node.      */


while ( depth[m] > depth[n] )
 {
  if ( up[m] != 1 )
   if ( flow [predl[m]] < fmin )
    {
     fmin = flow[predl[m]];
     nmin = m;
     cut = 2;
    }
   m = pred[m];
  }
 while ( depth[m] < depth[n] )
  {
   if ( up[n] == 1 )
    if ( flow[predl[n]] <= fmin )
     {
      fmin = flow[predl[n]];
      nmin = n;
      cut = 1;
     }
    n = pred[n];
   }

  /* Once you are at the same depth as the higher node move up the
```

```
        tree checking for the appropriate minimum flow on pred links
        until you come to the joining node of the cycle.          */

      while ( n != m )
       {
        if ( up[n] == 1 )
         if ( flow[predl[n]] <= fmin )
          {
           fmin = flow[predl[n]];
           nmin = n;
           cut = 1;
          }
        n = pred[n];
        if ( up[m] != 1 )
         if ( flow[predl[m]] < fmin )
          {
           fmin = flow[predl[m]];
           nmin = m;
           cut = 2;
          }
         m = pred[m];
        }
       jnode = m;
       linkout = predl[nmin];

if (level==6)
 printf ("Linkout is %d\n",linkout);

       flow[linkin] = fmin;
       n = from;
       m = to;

       /* Once we have decide which link leaves we update the flows on
          each link in the cycle mentioned above. By starting alterna-
          tively at the from and then the to node of the entering link
          we add the fmin flow to down links on the down side and sub-
          tract fmin from the up links on the upside. Then do the same
          thing for the predl of the pred node of  n and m until we come
          to the same node (joining node of the cycle). Finally set the
          flow of the entering link to fmin (done above actually )    */

       while ( n != jnode )
        {
```

```
        if ( up[n] == 1 )
         flow[predl[n]] -= fmin;
        else
         flow[predl[n]] += fmin;
        n =pred[n];

       }

      while ( m != jnode )
       {

        if ( up[m] == 1 )
         flow[predl[m]] += fmin;
        else
         flow[predl[m]] -= fmin;
        m = pred[m];

       }

/*****************************************************************************
*                                                                            *
*                      UPDATING AND REHANGING THE TREE                       *
*                                                                            *
*****************************************************************************/

      /* If the link leaving was on the downside A is the from node of
         the entering link and B is the to node else A is the to node
         and B is the from node. Then find C, D, E, F, l1, l2 (see var-
         iable glossary for definitions) and rehang the tree.        */


      if ( cut == 1 )
       {

         A = from;
         B =  to;

       }
      else
       {

         A = to;
         B = from;

        }

      flag2: C = pred[A];
      m = C;
      while ( m!= A )
       {

        last = m;
        m = thread[m];

       }

      F = last;
      l1 = linkin;
      l2 = predl[A];
```

```
oldepth = depth[A];
chdepth = depth[B] + 1 - oldepth;


/* Update the up variable of A    */


if ( cut ==1 )
 if ( up[A] == 0 )
  up[A] = 1;

if ( cut ==2 )
 if ( up[A] == 1 )
   up[A] = 0;


/* Determine the change in the dual prices  */


if ( up[A] == 1 )
 chdual = dual[B] - cost[l1] - dual[A];
else
 chdual = dual[B] + cost[l1] - dual[A];

m = A;

flag: last = m;


/* Update the dual and depth of A and its descendants  */


dual[m] +=  chdual;
depth[m] +=  chdepth;
m = thread[m];
if ( depth[m] > oldepth )
 goto flag;


/* Update the thread  */


D = last;
E = thread[D];
thread[F] = E;
thread[D] = thread[B];
thread[B] = A;



/* Update the pred and predl of A */


pred[A] = B;
predl[A] = l1;

/* If the predl of A is the link that's leaving we're done

   rehanging the tree. If not treat the predl of A like the

   link thats entering and repeat the process              */


if ( l2 != linkout )
 {
```

```
            linkin = 12;
            B = A;
            A = C;
            if ( B == bnode[linkin] )
             cut = 1;
            else
             cut = 2;

            goto flag2;

           }


            fmin = M;

      if   ( level==6 )

       {


      printf("Node Source Depth Thread Pred Predl Up     Dual\n\n");
      for (z=1; z<=tt; ++z )
       {

        printf ("%3d%6d%7d%7d%6d%5d%5d%8d\n",
                z,source[z],depth[z],thread[z],pred[z],predl[z],up[z],dual[z]);

        }

      }


/********************* Last Card of Pivot_out *********************************/

}
```

```
#include "macro.h"

/************** Body of Convert program ************************/

/***********************************************************************

Convert converts the double precision representation of job times
and lengths into an integer display

***********************************************************************/


convert (nn,ii)

/************* Declare Global Variables ************************/


double nn; /* the actual job start time or length */

int ii;  /* desired return parameter (month,day,hour,minute) */

{


/**************** Declare Local Variables ******************************/


int d1,d2,d3,d4,date[4];

double f4;

/* convert the month */
d1 = nn * 100000;
date[1] = d1/100000;

/* convert the hour */
d2 = d1 % 100000;
date[2] = d2/1000;

/* convert the minutes */
d3 = d2 % 1000;
date[3] = (d3*144)/60;

d4 = date[3] % 100;
f4 = (d4*1000);
f4 = f4/100000.0;
f4 = f4*60;

date[3] = date[3] - d4 + f4/1;

return (date[ii]);


/***************** Last card of Convert *********************************/

}
```

```
#include "macro.h"

/***************** Body of Assign program *********************************/
/*****************************************************************************
Assign actually assigns job vectors to a pilot based on the simplex solution
*****************************************************************************/



assign ()

{

/********************** Declare Global Variables *************************/

 extern JV job[],*job_pointer[];
 extern PV pilot[];
 extern int bflow[],bogus_label[],gold[],level,jobleft,q;
 extern int imp,bnode[],N,point[],flow[],cost[];

/******************** Declare Local Variables ****************************/

 int pp,trip=0,date[4],i,j=0,k,x,z; /* utility variables */
 int nflow=0; /* tracks bogus assigned arc flows */

 double stop; /* stop time of a job */



  jobleft = 0;
  if (level == 7)
    {
     printf ("\n\n");
     printf ("Nonzero flow for each link is:\n\n");
    }
  for ( i=1; i <= N; ++i )
   {
    if ( i != N )
     z = gold[i];
```

```
    else
     z = 0;

    for (k=point[i]; k<=point[i+1]-1; ++k )
      {

       bflow[k] = 0;



/* assign the job (bnode) the pilot (anode) */
       if ( flow[k] > 0 && imp != 1)
         {

          x = bnode[k] - N;
          j += 1;

          if (level==7)
           printf ("%6d.%2d%3d",i,x,flow[k]);

          while (flow[k] != 0 )
           {

            if ( x != q )
             {

              z += 1;
              pilot[i].assigned[z] = job[x];
              pilot[i].assigned[z].number = x;
              pilot[i].assigned[z].next = &pilot[i].assigned[z+1];
              gold[i] = z;

             }

           flow[k] -= 1;

           if ( i==N)
            nflow += 1;

          }


/* keep track of bogus assigned jobs with nflow          */
       if ( i==N && x != q  && bogus_label[pilot[i].assigned[z].number] != 1)
        {

         flow[k] = nflow;
         jobleft = 1;

        }

       nflow = 0;

       if ( j > 4 )
        {

         if (level==7)
         printf ("\n\n");

         j =0;

        }
```

```
    }

  }

}



/* formatting for printing the schedule */
  if(level==7)
    {

     printf ("\n\n");
     printf ("Pilot#    Job#    Start      Stop\n\n");

    }

   for ( i=1; i <= N; ++i )
    {

     while (job_pointer[i]->number != 0 )
      {

        for (pp=point[i]; pp<=point[i+1]-1; ++pp)
          {

           if (bnode[pp]-N == job_pointer[i]->number )
             bflow[pp] = 1;

          }

         job_pointer[i]= job_pointer[i]->next;

       }

      job_pointer[i] = &pilot[i].assigned[1];

    }

    if (level==7)
     for ( i=1; i <= N; ++i )
      {

       while (job_pointer[i]->number != 0 )
        {

         printf ("%3d%10d",i,job_pointer[i]->number);

         for (k=1; k<=3; ++k)
          date[k]=convert(job_pointer[i]->start,k);

         flug: if(date[3] <100)
          printf ("      00%2d",date[3]);

         else if ( date[3] < 1000 && date[3]>= 100 )
           printf ("      0%3d",date[3]);

         else
           printf ("      %4d",date[3]);

         if (trip != 1)
          {
```

```
            stop = job_pointer[i]->length + job_pointer[i]->start;
            for (k=1; k<=3; ++k)
             date[k]=convert(stop,k);
            trip = 1;
            goto flug;
           }
          else
           {
            trip = 0;
            printf("\n");
           }
          job_pointer[i] = job_pointer[i]->next;
         }
        job_pointer[i] = &pilot[i].assigned[1];
       }

/****************** Last Card of Assign ******************************/
     }
```

```
#include "macro.h"

/************** Body of Zero_pilotvector Program ******************/

/*****************************************************************************

This routine zero outs all assigned jobs to bogus in preparation for
the next transportation problem.

*****************************************************************************/

zero_pilotvector (y)

int  y;

{

/****************** Declare Global Variables ***************************/

    extern int N;

    extern PV pilot[];

    extern JV *job_pointer[];

    extern int gold[],bogus_label[];



/* zero out the bogus assigned jobs */

   while ( pilot[y].assigned[1].number != 0 )
    {

     if (pilot[y].assigned[1].label == 1)
      bogus_label[pilot[y].assigned[1].number] = 1;

     delete(y,pilot[y].assigned[1]);

    }

   job_pointer[y] = &pilot[y].assigned[1];


/************* Last Card of Zero_pilotvector ***************************/


}
```

```
#include "macro.h"

/********************** Body of Add Program ****************************/

/***************************************************************************
Add is a routine to add a job vector to a pilot's assigned list
***************************************************************************/

add(pil,vec)

/******************** Declare Global Variables *************************/

int pil;  /* the pilot to add the job to */

JV vec;  /* the job vector to add */

{


/************************ Declare Local Variables ************************/

int k,i;

extern PV pilot[];

extern int gold[];

i = pil; k=1;


/* find last assigned job and add new job after it */
while ( pilot[i].assigned[k].number != 0)
  {

   k += 1;
   pilot[i].assigned[k].next= &pilot[i].assigned[k+1];

  }

pilot[i].assigned[k] = vec;
pilot[i].assigned[k].next= &pilot[i].assigned[k+1];
gold[i] = k;


/********************** Last card of Add *******************************/
}
```

```
#include "macro.n"

/********************** Body of Delete Program ******************************/

/*****************************************************************************

Delete removes an assigned job from a pilot's list of jobs

*****************************************************************************/

delete(jock,jvec)


/********************** Declare Global Variables ****************************/

int jock; /* the pilot to remove the job from */

JV jvec;  /* the jobvector to remove from the above pilot */

{

/********************** Declare Local Variables *****************************/

int i,k,out;

extern int gold[],M;

i = jock; k=1; out = M;


/* find the given job and delete it */
for ( k=1; k<=gold[i]; ++k)
  {

   if ( pilot[i].assigned[k].number == jvec.number)
    out = k;

   if ( k >= out)
     {

        pilot[i].assigned[k] = pilot[i].assigned[k+1];
        pilot[i].assigned[k].next = &pilot[i].assigned[k+1];

      }

   }

  gold[i] -=1;


/*************************** Last card of delete ******************************/
}
```

```
#include "macro.h"

/******************* Body of Swap Program ******************************/

/***************************************************************************

Swap takes as an input a "bogus" assigned job and attempts to swap
it to a "real" pilot. If unable to do so the job is labeled to "bogus"
so that the algorithm may continue. For details of the algorithm see
section 2.2.3

***************************************************************************/



swap(p)

/* The entering parameter p is the pilot to swap jobs from (bogus) */

int p;

{

/************** Declare Global Variables ***************************/

    extern int swap_count,nogo,flow[],point[],gold[],level;

    extern int bnode[],q,N,M,feaslinks[];

    extern JV job[],*job_pointer[];

    extern PV pilot[];

/******************** Declare Local Variables ******************************/

    int repeatjcb; /* tracks whether this job ha been swapped out before */

    int skip; /* indicates to skip a pilot already swap[ped a job on this

    int track; /* indicates that an eligible swap pilot has been found */

    int noswap; /* general marker indicating no swap to a pilot */

    int  y,i,f,k,old=M,a,x,d,c,first,pj,sj;  /* utility */


    int bestswap[NJ]; /* indicates the best swap candidate to  a particular
                         job                                  */

    int oldcost; /* tracker for best cost */

/* local struct */
    JV olds;

 nogo = 1;

/* Lock at all bogus assigned jobs and swap them out   */
```

```
 while ( job_pointer[p]->number != 0 )
  {

   sj= job_pointer[p]->number;
   oldcost = 0; old = M; track =0; noswap = 0;

/* If the job is not the sink job and the job is not labeled swap it out */
   if ( sj != q && job_pointer[p]->label != 1)
    {

     if ( level==10)
       printf("We are swapping out job %d from pilot %d\n",
       job_pointer[p]->number,p);
     bestswap[sj] = p;

     for (i=1; i<N; ++i )
      {


       if ((noswap > 0 && track == 1  ) ||
          (track==0 && noswap==0  )      )

         {

          skip = 0;

          if ( skip == 0 )
           {


/* find the pilots qualified to do the bogus assigned job */
            f =1; k =1;
            while ( pilot[i].type[k] <= job[sj].type
                   && pilot[i].type[k] != 0)

              {

               if (pilot[i].type[k] == job[sj].type )
                 {

                  noswap = 0;
                  if ( level == 10 )
                   printf ("Pilot %d is qualified to do bogus job %d\n\n",
                   i,sj);

                  track =1;

/* of these pilots see which of their assigned jobs interfere with the
   bogus assigned job */
                  while ( job_pointer[i]->number != 0 )
                    if ( job_pointer[i]->number != q )
                     {

                      pj = job_pointer[i]->number;
                      y =1;
                      while ( job[sj].con[y] <= pj && job[sj].con[y] != 0)
                        {

                         if ( pj == job[sj].con[y])
                          {

                            if (level == 10 )
                              printf("Pilot %d's job %d interferes with the swap\n\r

/* as long as the intefering job(s) are not labeled to a pilot ( in which case
```

```
they cannot be swapped ) see if a feasible link ( an available pilot ) exists
for the interfering job */

                        if ( job_pointer[i]->label != 1)
                          {

                            if ( feaslinks[pj] != 1 )
                             {

                              noswap +=1;
                              if ( level ==10 )
                               {

                                 printf("In addition there is no one to take thi
                                 printf ("His cum noswap is %d\n",noswap);

                               }

                              }

                             }
                            else
                             {

                              noswap = M;
                              if ( level ==10 )
                              printf ("We can't swap with pilot %d since his job %

                             }

                            pilot[i].swapout[f] = pj;
                            pilot[i].swapout[f+1] = 0;
                            if ( level==10)
                            printf("If pilot %d does bogus job %d he'll have to dr
                            f += 1;

                           }

                          y +=1;

                         }

                        job_pointer[i] = job_pointer[i]->next;

                       }

/* Swap out pilots with the least number of interfering jobs. In case of ties
   select the pilot that has the best price for the bogus assigned job */
                      if ( noswap <= old && noswap<M)
                      {

                       if ( pilot[i].cost[k] < oldcost )
                        {

                         oldcost = pilot[i].cost[k];
                         old = noswap;
                         bestswap[sj] = i;
                         if ( level==10)
                         printf("The best pilot so far to swap job %d with is pil

                        }

                       }
                      else
                       {
```

```
                        if ( track == 1)
                         noswap = old;
                        else
                         noswap = 0;

                        if (level ==10)
                        printf("Can't swap pilot %d jobs with pilot %d\n\n",i,p);

                       }

                     }

                    k += 1;

                   }

                  job_pointer[i] = &pilot[i].assigned[1];

                 }

               }

              }

             }

  d = bestswap[sj];

/* if no one available to do the bogus assigned job label it to bogus */
  if ( d == p )
   {

    if (level==10)
     printf("We are labeling job %d to pilot %d\n\n",sj,
            d);
    job_pointer[p]->label = 1;

   }


/* otherwise label the job to the best candidate found */
 if ( d != p )
  {

   if (level==10)
    printf("We are swapping out with pilot %d\n\n",d);

   swap_count += 1;

   repeatjob=0; a = 1;
   while ( pilot[p].assigned[a].number != 0 )
    {

     if ( pilot[p].assigned[a].number == sj && repeatjob != 1)
       {

         pilot[p].assigned[a].label = 1;
         olds = pilot[p].assigned[a];
         delete(p,olds);
         add(d,olds);
         repeatjob = 1;

       }

      a += 1;
```

```
    }

   nogo = 0;
   c = 1; first = 0;

/* assign all of the intefering jobs to bogus */
   while ( pilot[d].swapout[c] != 0 )
     {

      if (level==10)
       printf("We are swapping out pilot %d job %d\n\n",d,pilot[d].swapout[c]);
      f = 1;
      while ( pilot[d].assigned[f].number != 0)
       {

        if (pilot[d].assigned[f].number == pilot[d].swapout[c])
         {

          olds =  pilot[d].assigned[f];
          delete(d,olds);
          add(p,olds);
          for(i= point[p]; i<point[p+1]; ++i)
             if(bnode[i]==N+olds.number)
               {
                flow[i] += 1;
                break;
               }
         }

        f += 1;

       }

      c += 1;

    }

   }

   job_pointer[p]= job_pointer[p]->next;

   }

  job_pointer[p] = &pilot[p].assigned[1];


/* adjust the flow to reflect the bogus assigned jobs */
  for ( x=point[p]; x<=point[p+1]-1; ++x )
   flow[x] = 0;

  while (job_pointer[p]->number != 0 )
   {

    flow[job_pointer[p]->number + point[p] -1] += 1;
    job_pointer[p] = job_pointer[p]->next;

   }

  job_pointer[p] = &pilot[p].assigned[1];

  if ( level == 10 )
   {

    for ( i=1; i <=N; ++i)
     {
```

```
      printf("\n\n");
      printf("Pilot %d's jobs are now:\n\n",i);
      while(job_pointer[i]->number != 0)
       {

        printf("%3d",job_pointer[i]->number);
        job_pointer[i] = job_pointer[i]->next;

       }

      job_pointer[i] = &pilot[i].assigned[1];

     }

    }


/************************** last card of swap ***************************/
}
```

```
#include "macro.h"

/************************ Body of Reheap Program ************************/

/*****************************************************************************

Reheap updates the "heap" vector following the reassignment of a job in
the improve phase.

*****************************************************************************/



reheap(jo,pi)

/********************** Declare Global Variables ************************/

int jo; /* the job number to reheap */

int pi; /* the pilot who now has the job */

{

/********************** Declare Global Variables ************************/


extern PV pilot[];
extern JV job[];
extern HP heap[], *hp;
extern int point[],bnode[],cost[],N;


/********************** Declare Local Variables ************************/
int jj,d,k;
/* initialize */
  jj = jo;
   d = pi;
  heap[jj].pilot = d;


/* assign new arc to the heap struct */
  for (k=point[d]; k<=point[d+1]-1; ++k)
      if (bnode[k]-N == jj)
        {

         heap[jj].arc = k;
         break;

        }


/* assign new cost to the heap struct */
  for (k=1; k<=pilot[d].next; ++k)
    if (pilot[d].type[k]==job[jj].type)
      {

         heap[jj].cost = pilot[d].cost[k];
         break;

      }

    }
/************************ Last card of Reheap ***************************/
```

```
#include "macro.h"

/************************** Body of Impr Program **************************/

/*****************************************************************************

Impr is the routine to improve the best ( hopefully feasible ) solution found
as of yet by the simplex and swap routines. The modified two opt procedure
used is described in section 2.5 .

*****************************************************************************/

impr(jj,pp,cs)

/************************** Declare Global Variables **********************/

int jj; /* the job we seek to improve the price of */

int pp;  /* the pilot currently assigned the job */

int cs; /* the current price gained for a job */

{


/************************** Declare Global Variables **********************/

extern int imp_count,q,level,N,imp;
extern PV pilot[];
extern JV job[],*job_pointer[];


/************************** Declare Local Variables ***********************/

int pj,i,j,k,n,m; /* utility variables */

int chcost; /* the change in cost with a given swap */

int bestcost; /* the bestcost found for a given swap */

int noswap; /* indicates that noswap is possible */

int jobswap[ND],swjb[ND]; /* both indicate the jobs that must swap to
                             complete an improve swap   */

int bestswap[NJ]; /* indicates the best pilot to swap for */

int pilot_check; /*  marker indicating a pilot has been checked for a swap */

int cumswap; /* the cumulative swap number for a given improvement */



/* struct */
JV jvc;


/* initialize parameters  */

for (i =0; i<ND; ++i)
 {
    jobswap[i] = 0;
    swjb[i] = 0;
```

```
 }
chcost = 0;
bestcost = 0;
bestswap[jj] = pp;


/* find each pilot qualified to do a given job */
for (i=1; i<N; ++i)
 {

  cumswap = 0;
  noswap = 0;
  pilot_check = 0;

  for (j=1; j<=pilot[i].next; ++j)
    if ( pilot[i].type[j] == job[jj].type )
        {

         pilot_check = 1;
         if ( level == 10 )
         printf ("Pilot %d is qualified to do job %d\n\n", i, jj);

/* for each pilot qualified to do a given job see which of his assigned jobs
interfere with it                                                    */
         while ( job_pointer[i]->number != 0 )
         if ( job_pointer[i]->number != q)
           {

            k =1;
            pj = job_pointer[i]->number;
            while ( job[jj].con[k] <= pj && job[jj].con[k] != 0 )
              {

               if ( job[jj].con[k] == pj )
                 {

                  cumswap += 1;
                  jobswap[cumswap] = pj;
                  if ( level == 10 )
                   printf (" Pilot %d's job %d interferes with job %d\n\n"
,i, pj, jj);


                 }

                k += 1;

               }

              job_pointer[i] = job_pointer[i]->next;

             }

          job_pointer[i] = &pilot[i].assigned[1];


/* see if the pilot who is dropping the improve job can pich up all of these
   interfering jobs                                                  */
        while ( job_pointer[pp]->number != 0 )
        if ( job_pointer[pp]->number != q )
          {

           pj = job_pointer[pp]->number;
           if ( pj != jj )
             {
```

```
                n=1;
                while ( jobswap[n] != 0 )
                  {

                    k =1;
                    m = jobswap[n];
                    while ( job[pj].con[k] <= m && job[pj].con[k] != 0 )
                      {

                       if ( job[pj].con[k] == m )
                        {
                        if ( level == 10)
                         printf("TOO BAD, pilot %d can't do pilot %d's job %d, there
                        noswap = 1;

                        }

                        k += 1;

                       }

                     n += 1;

                    }

                  }

                job_pointer[pp] = job_pointer[pp]->next;

               }

             job_pointer[pp] = &pilot[pp].assigned[1];


/* if he can, see if this will improve the objective, if so keep track of it */
        if ( noswap != 1 )
          {

            chcost = 0;
            n = 1;
            while ( jobswap[n]  != 0 )
              {

               m = jobswap[n];
               for (k=1; k<=pilot[i].next; ++k )
                 if ( pilot[i].type[k] ==  job[m].type)
                  {

                    chcost -= pilot[i].cost[k];
                    break;

                  }
               for (k=1; k<=pilot[pp].next; ++k )
                 if ( pilot[pp].type[k] ==  job[m].type)
                   {

                    chcost += pilot[pp].cost[k];
                    break;

                    }

               n += 1;

              }

            chcost += (pilot[i].cost[j]-cs);
```

```
            if ( level == 10 )
            printf ("The chcost if pilot %d picks up job %d is %d\n\n",
i,jj,chcost);

            if ( chcost < bestcost )
              {

               chcost = bestcost;
               bestswap[jj] = i;
               n =1;
               while ( jobswap[n] != 0 )
                 {

                  swjb[n] = jobswap[n];
                  n += 1;

                 }

               while ( swjb[n] != 0)
                 {

                   swjb[n] = 0;
                   n+=1;

                  }

              }


          }


       if (pilot_check == 1)
         j = pilot[i].next;

     }

n=1;
while ( jobswap[n] != 0 )
   {

     jobswap[n] = 0;
     n += 1;

    }
}

/* swap out with the best changecost */

if (bestswap[jj] != pp )
  {

   imp_count += 1;

   i = bestswap[jj];
   if ( level == 10 )
   printf("We're swapping out with pilot %d\n\n", i);

   n=1;
   while ( swjb[n] != 0 )

     {
      m = 1;
      while (pilot[i].assigned[m].number != swjb[n] && m < NJ)
```

```
        m += 1;
        jvc = pilot[i].assigned[m];
        delete(i,jvc);
        add(pp,jvc);
        reheap(swjb[n],pp);
        n += 1;

       }

     m = 1;
     while ( pilot[pp].assigned[m].number != jj  && m < NJ)
      m += 1;
     jvc = pilot[pp].assigned[m];
     add(i,jvc);
     reheap(jj,i);
     delete(pp,jvc);
     imp = 1;

    }
else

   {

    if ( level == 10 )

     printf ("No good swaps for job %d\n\n",jj);

    }

/********************** Last card of Impr **********************************/
}
```

END

FEB.

1988

DTIC